AFRL-RI-RS-TR-2011-174

SEAMLESS AND SECURE FEDERATION AMONG HIGHLY AND LOOSELY CONNECTED INFOSPACES (INFOFED)

FLORIDA INSTITUTE FOR HUMAN & MACHINE COGNITION (IHMC)

JULY 2011

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2011-174 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/
ASHER SINCLAIR
Work Unit Manager

/s/
JULIE BRICHACEK, Chief
Information Systems Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JUL 2011	Final Technical Report	MAY 2007 – NOV 2010

4. TITLE AND SUBTITLE

SEAMLESS AND SECURE FEDERATION AMONG HIGHLY AND LOOSELY CONNECTED INFOSPACES (INFOFED)

5a. CONTRACT NUMBER
FA8750-07-2-0174

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62702F

6. AUTHOR(S)

Niranjan Suri, Andrzej Uszok

5d. PROJECT NUMBER
558J

5e. TASK NUMBER
07

5f. WORK UNIT NUMBER
02

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Florida Institute for Human & Machine Cognition (IHMC)
40 S. Alcaniz Street
Pensacola FL 32502

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RISE
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2011-174

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. PA# 88ABW-2011-3478
Date Cleared: 15 Jun 11

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report describes research conducted toward developing a federated approach to interconnecting multiple information spaces to enable data interchange. We propose a set of interfaces to facilitate dynamic, runtime discovery and federation of information spaces. We also report on integrating with the KAoS policy and domain services framework to realize policy-based control over the federation and exchange of information. Our approach allows clients to transparently perform publish, subscribe, and query operations across all the federated information spaces. We have integrated with three existing JBI implementations – Apollo from the Air Force Research Laboratory, Mercury from General Dynamics and AIMS (Advanced Information Management System) from Northrop Grumman. Both Mercury and AIMS are independent implementations that comply with the JBI architecture. Most recently, we have integrated with Phoenix, a fully SoA (Service-oriented Architecture) based approach to information management. As part of this effort, we identified changes that needed to be made to the Phoenix architecture and implementation to support federation, and refactored the federation capabilities into a set of services that can be enabled on demand within Phoenix.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			ASHER D. SINCLAIR
U	U	U	UU	73	19b. TELEPHONE NUMBER (*Include area code*) N/A

Table of Contents

Summary 1
1. Introduction 2
2. Methods, Assumptions, and Procedures 2
3. Results and Discussion 3
3.1. Design, Architecture, and Implementation 4
3.1.1 Approach for CAPI-based Infospaces 6
3.1.2 Approach for Apollo 12
3.1.3 Initial Approach for Mercury 13
3.1.4 Evolving Approach for Phoenix 13
3.2 Discovery 16
3.3 Monitoring 17
3.4 Policies and Contracts 20
3.4.1 Technical Overview of the KAoS Services Framework 21
3.4.2 Controlling Federation 23
3.4.3 Contract Negotation Example 28
3.5 Adaptation 30
3.5.1 Federation Adaptation Service Components 33
3.5.2 Network Monitoring Using Mockets 37
3.5.3 State Memory Algorithm 38
3.6 Performance Evaluation 42
3.6.1 Experiments with Apollo and Federation 42
3.6.2 Experiments with Apollo, Phoenix, and Federation 47
3.6.3 Experiments with Phoenix and Federation 49
4. Conclusions 63
5. Recommendations 63
5.1 Extensions to Current Federation Capabilities 64
5.2 Integration with Other Systems and Frameworks 64
5.3 Enhancement of Transport and Dissemination Channels 65
5.4 Integration, Evaluation, and Experimentation 65
6. References 66
List of Symbols, Abbreviations, and Acronyms 67

List of Figures

Figure 1: Architecture of a JBI-Oriented Information Management System 3
Figure 2: Example federation between Federates A, B and C 5
Figure 3: Federation Services inside CAPI-based infospace architecture 7
Figure 4: Federation Interfaces and Federation Service Architecture 9
Figure 5: Details of the Federation Service interfaces for CAPI based information spaces 10
Figure 6: Federation query workflow 11
Figure 7: Federation Service components inside the Apollo Architecture 13
Figure 8: Final architecture of the Federation Service integration into Phoenix 14
Figure 9: Monitoring Service for Phoenix 18
Figure 10: Updating and Notification of Metrics in the Monitoring Service 20
Figure 11: KAoS Policy Services Conceptual Architecture 22
Figure 12: KAoS Guard – the policy decision point integrated with the Federation Service 23
Figure 13: Federation Metadata GUI in KPAT 25
Figure 14: Contract GUI in KPAT 25
Figure 15: KPAT configuration for the Federation Service Policy 27
Figure 16: Policy Wizard for the definition of federation policies 28
Figure 17: Contract Example Details 29
Figure 18: Components and Connections Related to Federation Adaptation Service 35
Figure 19: UML Diagram of Classes Used in the State Memory Algorithm 40
Figure 20: Configuration to Measure Baseline Performance of Apollo 43
Figure 21: Configuration to Measure Performance of Two Federates with Apollo 43
Figure 22: Configuration to Measure Performance of Three Federates with Apollo 43
Figure 23: Configuration to Measure Latency with Baseline Apollo 44
Figure 24: Configuration to Measure Latency with Two Federates and Apollo 44
Figure 25: Publisher Performance with Apollo and Federation 45
Figure 26: Subscriber Performance with Apollo and Federation 46
Figure 27: Latency of Information Delivery with Apollo and Federation 47
Figure 28: Experimental Scenario for the Performance Evaluation: The Baseline Version of the Tested IMS is shown in A. B shows the Configuration for the Tests with the Federation Service 48
Figure 29: Comparison of Publication Time with 0 KB Payload 52
Figure 30: Comparison of Publication Time with 10 KB Payload 52
Figure 31: Comparison of Publication Time with 100 KB Payload 52
Figure 32: Comparison of Publication Rate with 0 KB Payload 53
Figure 33: Comparison of Publication Rate with 10 KB Payload 53
Figure 34: Comparison of Publication Rate with 100 KB Payload 53
Figure 35: Comparison of Subscriber Time with 0 KB Payload 54
Figure 36: Comparison of Subscriber Time with 10 KB Payload 54
Figure 37: Comparison of Subscriber Time with 100 KB Payload 55
Figure 38: Comparison of Subscription Rate with 0 KB Payload 55
Figure 39: Comparison of Subscription Rate with 10 KB Payload 56

Figure 40: Comparison of Subscription Rate with 100 KB Payload 56
Figure 41: Average Latency of Objects Received by Subscribers with 0 KB Payload ... 57
Figure 42: Average Latency of Objects Received by Subscribers with 0 KB Payload ... 57
Figure 43: Average Latency of Objects Received by Subscribers with 10 KB Payload . 58
Figure 44: Average Latency of Objects Received by Subscribers with 100 KB Payload 58
Figure 45: Bandwidth Comparison (Bytes) of Baseline Phoenix and Federation 60
Figure 46: Bandwidth Comparison (Packets) of Baseline Phoenix and Federation 61
Figure 47: Performance (Time) Comparison for Publisher with Varying Channel Capacities 62
Figure 48: Performance (Rate) Comparison for Publisher with Varying Channel Capacities 62
Figure 49: Performance (Time) Comparison for Subscribers with Varying Channel Capacities 63
Figure 50: Performance (Rate) Comparison for Subscribers with Varying Channel Capacities 63

Summary

Network-centric warfare is a cornerstone of modern warfighting. Timely access to relevant data and information is critical to successful mission execution in network centric warfare. Often, the data required to support a mission is not always produced or resident within a single system, but is distributed among multiple systems that must be dynamically interconnected to support the overall data and information needs.

While proprietary and stove-piped information systems have slowly given way to standardized information management architectures (such as the Joint Battlespace Infosphere (JBI) architecture developed by the US Air Force Research Laboratory), each independent organization and/or mission is normally associated with a separate instance of a managed information space that operates in an independent manner. This is necessary given the different stakeholders and administrative domains responsible for the information. However, the demands for coordination and cooperation require interoperability and information exchange between these independently operating information spaces.

This report describes research conducted toward developing a federated approach to interconnecting multiple information spaces to enable data interchange. We propose a set of interfaces to facilitate dynamic, runtime discovery and federation of information spaces. We also report on integrating with the KAoS policy and domain services framework to realize policy-based control over the federation and exchange of information. Our approach allows clients to transparently perform publish, subscribe, and query operations across all the federated information spaces. We have integrated with three existing JBI implementations – Apollo from the Air Force Research Laboratory, Mercury from General Dynamics and AIMS (Advanced Information Management System) from Northrop Grumman. Both Mercury and AIMS are independent implementations that comply with the JBI architecture. Most recently, we have integrated with Phoenix, a fully SoA (Service-oriented Architecture) based approach to information management. As part of this effort, we identified changes that needed to be made to the Phoenix architecture and implementation to support federation, and refactored the federation capabilities into a set of services that can be enabled on demand within Phoenix.

We also report on discovery mechanisms that were developed and integrated to facilitate discovery of potential federates to form a federation, a novel transport protocol based on the Mockets communications library, and an adaptation mechanism that dynamically modifies the behavior of the federation in order to maintain desired quality of service properties.

During the course of the project, several experiments were conducted to measure the performance and overhead of federation, which are reported in this document. We also report on numerous experiments conducted with the Mockets communications library in the context of the JEFX (Joint Expeditionary Force Experiment) 2010.

As a result of this project, we have shown the feasibility of federation, as well as empirically measured the performance of a reference implementation for federation. We have identified and provided solutions for key requirements, such as policy-based control, discovery, transport protocols, and adaptation capabilities.

1. Introduction

Information systems are a key component of any military mission and are essential to ensuring their successful execution. Traditionally, information management was supported by stove-piped systems that were difficult to update, modify, and integrate. In order to address this problem, the US Air Force Research Laboratory developed the Joint Battlespace Infosphere (JBI) architecture [1] first and started working on the Phoenix specification [2] afterwards. Both JBI and Phoenix try to define a standard for the implementation of information management architectures that support a publish/subscribe/query model. In addition to that, the JBI architecture standardizes the interfaces for client applications (CAPI – the Client API) to facilitate client integration into any JBI implementation.

This standardization enables the implementation of information management architectures that are based on a common information management model. However, the interconnection and information sharing between information spaces (infospaces) that may belong to different administrative domains still remains an open issue.

Federation solves this problem by supporting the interconnection of multiple, independently managed infospaces for information sharing. This report describes our approach to federation, in the context of the Apollo and Mercury implementations of JBI, and then in the context of the Phoenix services-based approach to JBI. We propose a set of interfaces and services to facilitate dynamic, runtime discovery and federation of infospaces. We integrate with KAoS - policy and domain services framework, to realize policy-based control over the federation and the exchange of information. We also describe our approaches to discovery, monitoring, and adaptation. Our approach allows clients to transparently perform publish, subscribe, and query operations across all the federated information spaces.

2. Methods, Assumptions, and Procedures

The primary method applied to this project was a spiral design-implement-evaluate process, which resulted in several iterations and implementations, with each implementation coming closer to satisfying the needs of federating multiple information spaces.

The assumptions we make are that the federation capability is designed in particular for JBI-style information management systems. The architecture and motivations for JBI are described in detail in [3], which presents a reference model for information management. The elements of the JBI architecture essential for the scope of federation are highlighted in Figure 1.

An Information Space is defined as one instance of a JBI based system, which facilitates exchange of information between clients. A number of clients connect to the system, behaving as producers and/or consumers of information.

The system includes both an Information Catalogue, that is a directory of information types known to the system, as well as an Information Repository, which handles the actual data. The Information Repository may optionally archive information for later retrieval using queries. Different JBI based implementations are free to use any approach as long as they comply with the syntax and semantics of the CAPI - the Client API. In the case of Apollo, one of AFRL's

reference implementations of the JBI information management concepts, the Information Catalogue is called the Metadata Repository (MDR), while the Information Repository is called the Information Object Repository (IOR). Published data is represented as a Managed Information Object (MIO). Each MIO has a corresponding data type that is registered in the MDR, metadata in the form of an XML document, and a payload. Clients may have standing subscriptions based on the type, with an optional predicate to match against published metadata. If a predicate is specified, it is in the form of an XPATH expression, which can filter out unnecessary MIOs that a client is not interested in receiving. Clients may also execute queries that result in matching MIOs being retrieved from the IOR and returned to the client.

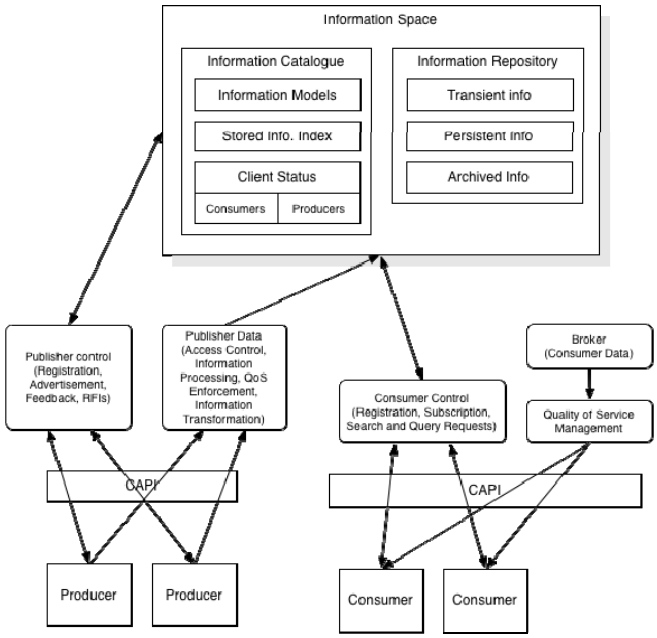

Figure 1: Architecture of a JBI-Oriented Information Management System

A client typically connects to one (and only one) Information Space. While it is possible to connect to multiple information spaces, doing so places the onus on the client to discover the information spaces and connect to each one. The client would also need to be authenticated with multiple information spaces, which implies that all of them must have accounts for the client (difficult when there are multiple administrative domains involved). One of the benefits of Federation is to make the presence of multiple information spaces transparent to the clients. A client does not need to know the network endpoints of multiple information spaces to attach to. Each client continues to connect to one information space, but has access to all allowed information (controlled by policy) across multiple information spaces.

3. Results and Discussion

This section presents and discusses the results of the project. The results are divided into two distinct kinds. Firstly, we present the result of our iterative design process, and describe the federation architecture that was developed initially for Apollo and Mercury, and then redesigned for Phoenix. Secondly, we describe other key components of the federation capability, including the KAoS Policy and Domain Services components, the Discovery components, the Monitoring

components, and the Adaptation components. Finally, we present detailed experimental results obtained by using these implementations in specific scenarios.

3.1. Design, Architecture, and Implementation

The goal of federation architecture has been to supports seamless and secure integration of multiple information spaces, each of which is called a federate. Seamless implies that the architecture supports automatic discovery of and interconnection between federates. The process of federation is transparent to clients, which still connect to their home federate as normal. Secure implies that the federation process is not arbitrary and open. The establishment of federation and exchange of information is controlled via policies.

One important aspect of our federation architecture is that all federates are peers. Each federate independently manages its connection with other federates. Each federate has its own set of policies that govern the exchange of information with other federates. This approach is logical given that each federate could potentially be in a separate administrative domain.

Primary goal of this research effort was the development of a generic set of services (and their interfaces) supporting federation in order to obtain a flexible architecture easily adaptable to different IMS implementations. After examining both the legacy JBI CAPI specification and a number of its implementations (i.e. Apollo from AFRL, Mercury from General Dynamics and AIMS from Northrop Grumman) as well as the most recently released service-based specification, Phoenix and its initial release, we came out with the below set of federation services, which will be in detail described in the subsequent part of the report:

- Discovery Manager (DM) provides the discovery functionalities that are necessary to automatically find other potential federates in the network.
- Federation Manager (FM) takes the necessary actions when new potential federates are discovered by the DM and when connections with remote federates are terminated.
- Remote Federation Service Proxy (RFSP) responsible for interception of local subscription, publication and queries and forwarding them to the remote federate.
- Federation Monitoring Component (FMC) implements the functionalities for monitoring the behavior of Federation services.
- Federation Policy and Contract Service which controls federation through policies and negotiates contracts with new federates.
- Adaptation Manager (AM) takes advantage of the application-level statistics along with information about system and network behavior to dynamically adapt the behavior of federation.

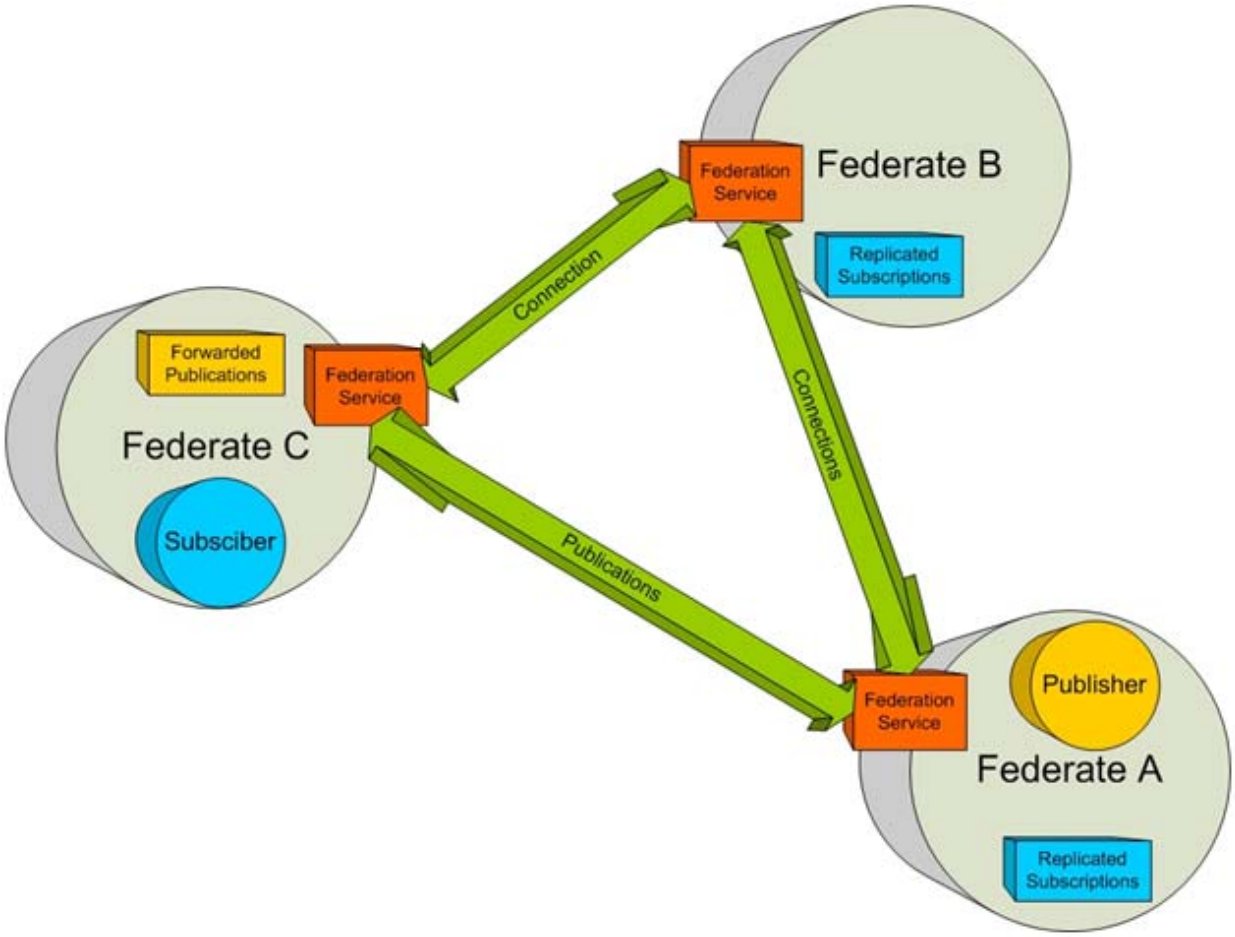

Figure 2: Example federation between Federates A, B and C

At this point, we generally describe how the functionality of federation is being handled by these designed and developed services. For simplicity, we will consider a scenario where the federation happens between three instances of an Information Management System (IMS), which we will refer to as Federate A, Federate B and Federate C (Figure 2). We will also assume that the nodes where the IMSs run are discovered with a lower level discovery-enabled communication substrate. The specific discovery approaches supported by our Federation implementation are described later. From the perspective of the Federation Service, we assume that each instance (each Federate) has created a Server Socket (using a configured port on the host system that it has been installed). This endpoint (a combination of the IP address of the host and the port being used by the Federation Service) is advertized to the underlying discovery capability. This discovery process then provides the endpoint address (IP address and port) for each federate to the other, as they become available (that is, visible and reachable across a network).

Federation Establishment

When the Federation Service is instantiated along with the other services that are part of the IMS architecture, the first step is the registration by the Discovery Manager with the discovery and grouping API provided by the sub-layer with such capabilities. By registering and joining a predefined group, the IMS manifests its intention of being part of the federation. Once that

happens, each IMS instance is notified about the presence of the other. At this point, a handshake phase starts. During the handshake, each potential federate opens a connection to the other and eventually a contract negotiation occurs. Upon contract acceptance by both nodes, the federation is officially established, and each federate creates an instance of a Remote Federation Service Proxy. At this point existing subscriptions are exchanged by federates.

Subscription forwarding
When a client connected to Federate A issues a subscription with its local IMS, the request is intercepted by the RFSPs for B and C which forward the received request for subscription to its remote federates. Once Federate B and C obtain it, the subscription is stored in a remote subscriptions table, ready to be matched against local publications.

Publication handling
When a client publishes information to the local IMS (Federate A), such publication is intercepted by the RFSPs for B and C. In normal conditions (i.e., with no adaptations in effect), Federate A attempts to execute the predicate matching locally, by comparing the publication type and metadata with the remote subscriptions it may have previously stored in its remote subscription table. Publications for which the local matching succeeds are marked as matched, and sent to Federate B via the RFSP. Federate B receives the publication, verifies if it was already matched (and if it was not it matches it with the local subscriptions), and forwards it to the IMS. Finally the IMS takes care of the delivery to the correct subscriber clients.

Query handling
When a client queries a local IMS (Federate A), the query is intercepted by the RFSP for B and C. They forward the query to the remote federates and then wait for the possible query results from them. If they receive any results then they combine them with the local results delivered to the client.

Federation termination
Federation lasts until at least one of the nodes dies or leaves the federation group. When the other is notified about one of these events, it cleans up any references to the former remote federate, including any cached remote subscriptions.

Policies and contract
All the federation operational behavior detailed above is entirely governed by policies. Before performing any step in its execution flow, the FS verifies with the policy framework whether the current operation is allowed, and whether there are any restrictions to be imposed.

3.1.1 Approach for CAPI-based Infospaces

Figure 3 below shows federation service components inside the CAPI-based infospace. The shaded boxes represent new components that have been added to the original architecture for an information space. The three major components are: a *Federation Service* (FS), a *Federation Connector* (FC), and *Transformation Components* (TC). Each federate has one instance of an FS. Each federate also has n-1 instances of FCs, where n is the number of other federates that are

part of the federation. That is, each FC instance handles the connection to one remote infospace. The TCs are deployed as needed based on policy requirements.

The *Federation Service* (FS) handles the problem of *redirection*, as it manages data exchange between the infospace and its federates. As a prerequisite for interaction with federates, the manager of a given infosphere can configure its FS with information about other infospaces. This can include a set of policies that specify obligations and constraints on the behavior of the FS.

Note that the FS behaves both in the role of a consumer and producer with respect to other FS instances. In the role of a consumer, the FS will forward queries and predicates to receive remote MIOs whereas in the role of a producer, the FS will forward advertisements and locally produced MIOs to remote infospaces.

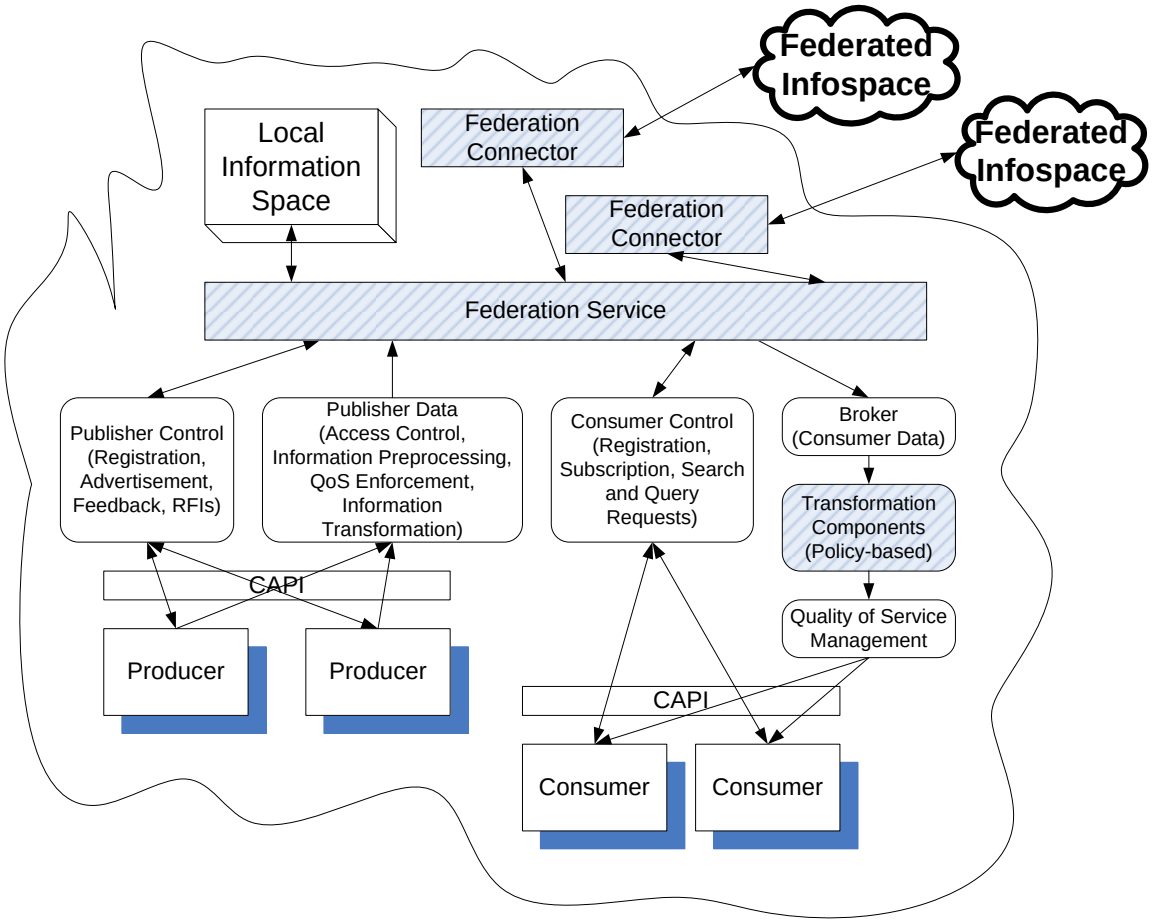

Figure 3: Federation Services inside CAPI-based infospace architecture

After examining the JBI CAPI architecture and its different implementations, Apollo from AFRL and Mercury from General Dynamics, we developed the following five key interfaces: *IMDService*, *InfoObjectReceptor*, *QueryReceptor*, *PredicateEvaluator*, and *AdaptationOracle*. These interfaces are implemented by various classes as described below. Figure 4 and Figure 5 below show the important components inside the *Federation Service*.

The IMDService supports all of the operations that one federate may want to execute on another federate. It is implemented by the *Federation Service* (FS) and is invoked by the local implementation of the information management system (IMS) – for example, Apollo. Each remote federate is represented by an instance of a *Remote Federation Service Proxy* (RFSP), which also implements the IMDService interface. Each proxy contains an instance of a *Federation Connector* (FC) and an instance of a *Remote Request Handler* (RRH), both of which also implement the same interface. The FC handles the network communication with the remote federate. The RRH receives incoming requests from the remote federate and executes them on the local IMS.

Consider the example of handling a new publication from a client. This results in the local information space invoking *newPublication*() on the FS. The FS invokes *newPublication*() on each of the active RFSP. If allowed by policy, and if the publication matches a remote subscription, the RFSP invokes *newPublication*() on the FC, which serializes and transmits the published object to the remote federate. The RFSP therefore acts as a Policy Enforcement Point (PEP). On the remote side, the FC receives the object and passes it to the RRH, using the newPublication() method again. The RRH on the remote side then injects the published object into the remote information space, where it is delivered to any relevant clients. Information objects are delivered via the *InfoObjectReceptor* interface, which is implemented by a modification made to the remote information space implementation. The process is inverted when a client of a remote federated publishes an information object that is received by the local federate.

In the case of a remote federate invoking a query, the query is executed by invoking the local IMS via the *QueryReceptor* interface. While not shown in the figure, the *PredicateEvaluator* interface is used when a predicate for a remote subscription needs to be evaluated. This is invoked by the RFSP, when a new publication is received, in the cases where a remote subscription has a predicate.

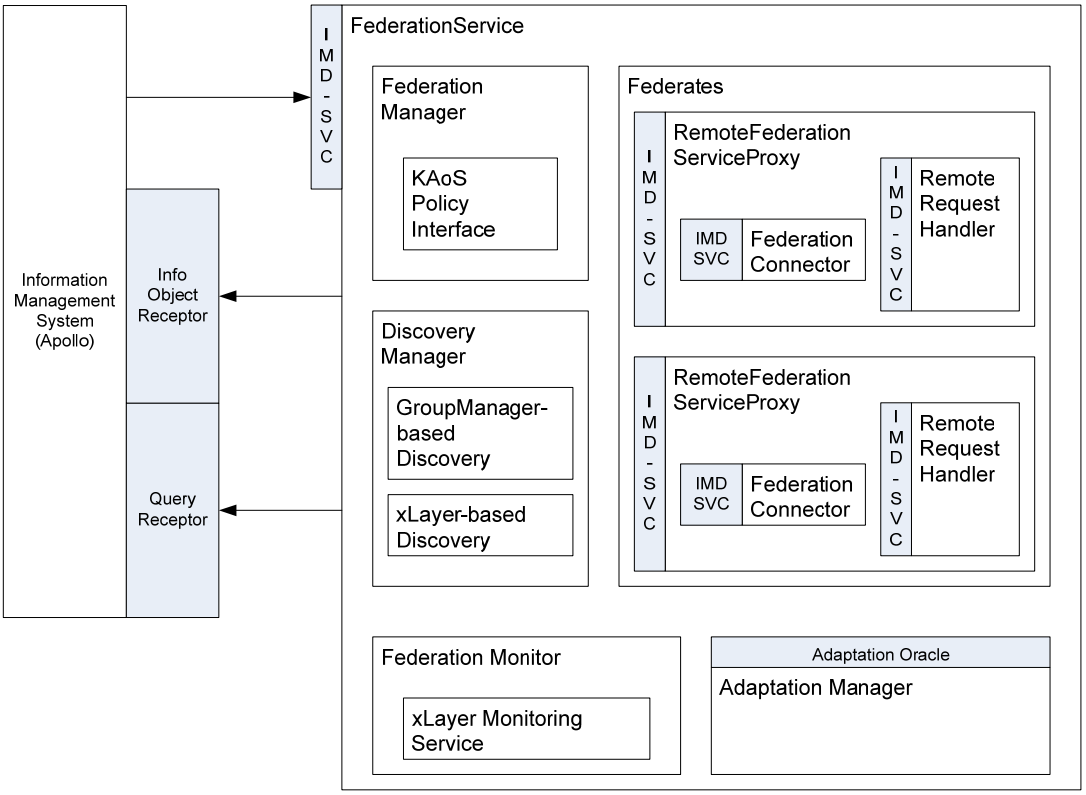

Figure 4: Federation Interfaces and Federation Service Architecture

The other major components shown in the figure are the *Discovery Manager* (DM) and the *Federation Manager* (FM). The DM handles the discovery of remote federates using two options – one based on the Group Manager and the other based on the XLayer cross-layer substrate. The XLayer substrate also provides a monitoring service that maintains detailed statistics and trends regarding the behavior of the network as well as the federation. For example, statistics such as CPU load, bandwidth utilized per connection to each remote federate, and the hit rate of remote predicates.

The final important component in the architecture is the *Adaptation Manager* (AM) and the *AdaptationOracle* interface. The AM automatically and dynamically changes the behavior of the federation to adapt to changing runtime conditions. The other components in the FS consult the AM via the AdaptationOracle interface.

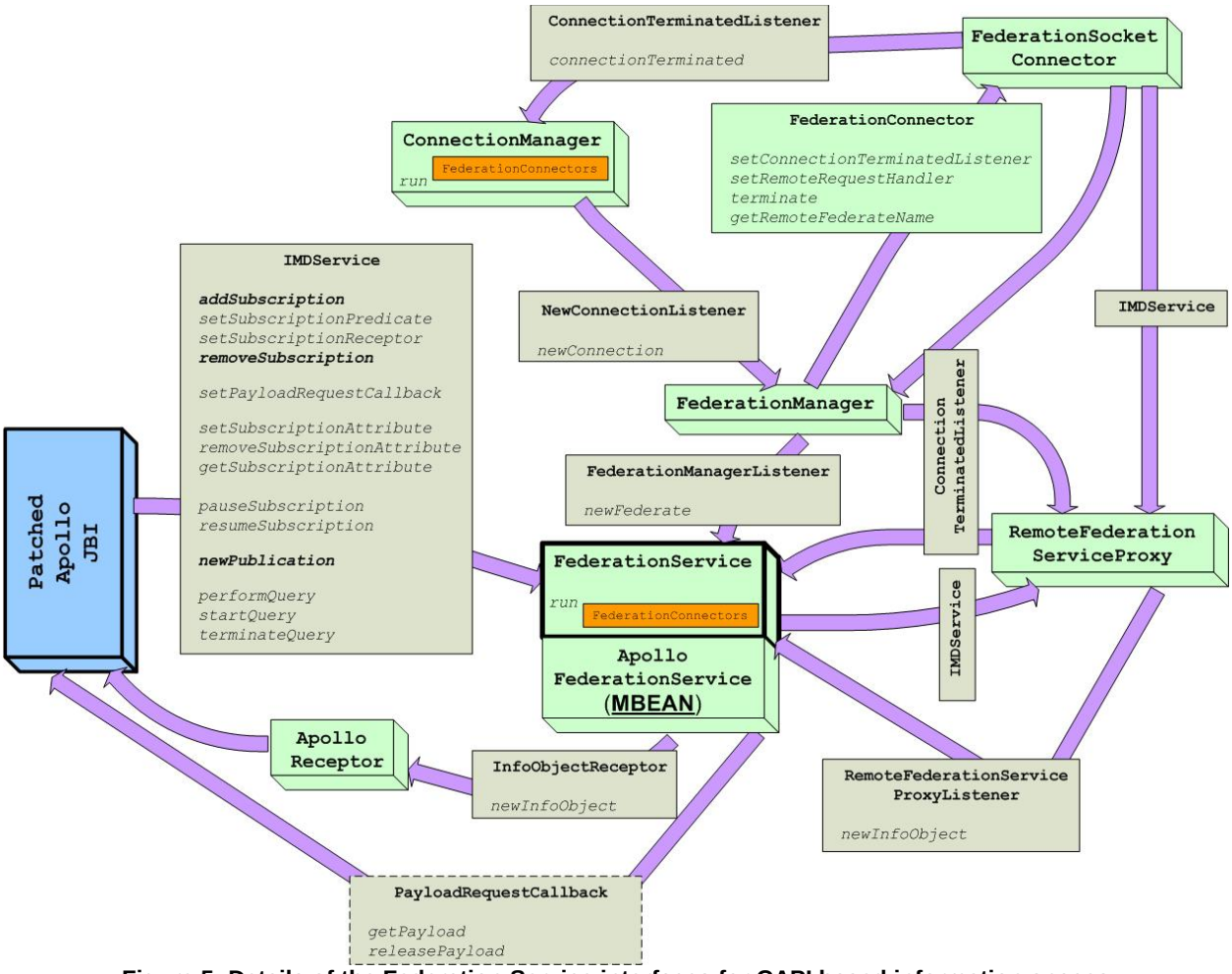

Figure 5: Details of the Federation Service interfaces for CAPI based information spaces

Federation Query

Queries are first satisfied using the local IMS. However if the federation was established then queries are forwarded to all federates allowed by policies. The remote federate attempts to report expected number of query results first to the querying federate and then actual queries. The remote query results are stored and used when the local results are exhausted. If there are no remote query results, the system times out after waiting for a configured length of time. Remote results can be archived locally in order to serve subsequent queries from local cache, dependent on policy settings. The federation query mechanism ensures that MIOs are not replicated when reporting remote results. Because of the distributed nature of the federation queries and the uncertainty of federate response times, we've introduced a default wait time for federate results. Clients can use a query sequence attribute to specify its specific timeout: *FederationQueryTimeOut* (default 5000 ms). Waiting for the whole timeout period is not required if:

- The query is not sent to any federate (because there are not federates due to prevention by policy),
- The query sent to all federates reports 0 results,
- The federates already returned the number of results they originally reported,

This information is sent to Query Receptor, so it interrupts its wait if no query results are expected.

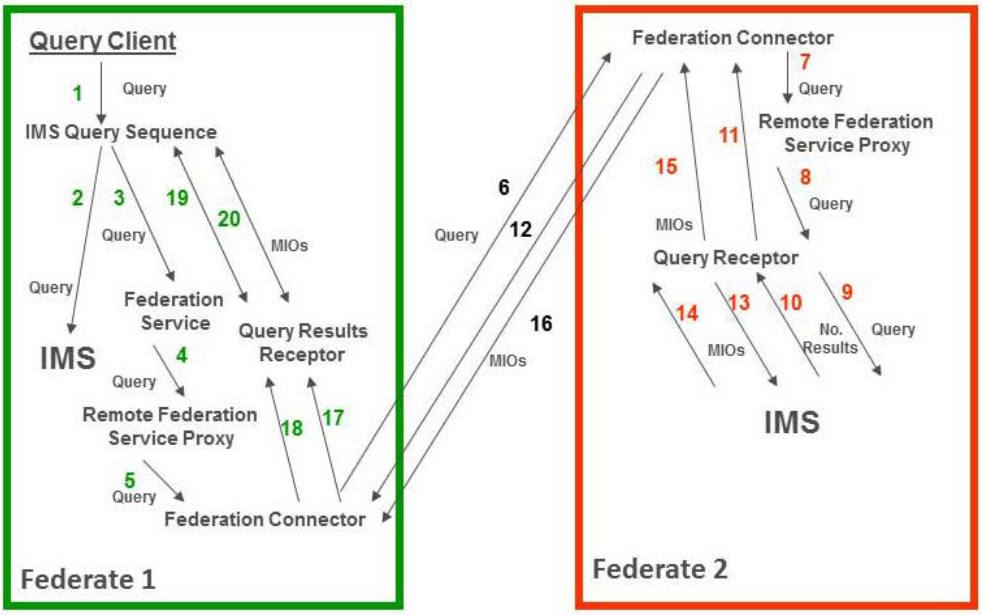

Figure 6: Federation query workflow

The query workflow presented on Figure 6 begins when the client issues the query. The query it sent to the local IMS query Service (step 1). This service has been modified and includes the Query Receptor. Thus the query is sent both to the local IMS (step 2) and to the Federation Service (step 3). Next the query is sent to the Remote Federation Service Proxy (step 4), which consults the policy how to handle the query and next to the Federation Connector (step 5). The connector forwards the query to the remote federate (step 6). The remote Federation Connector sends the query to the Remote Federation Service Proxy (step 7) which issues the query to the local IMS (step 8 and 9). The local IMS first reports the number of results for the query (step 10) to the federation Query Receptor, which forwards it through the Federation Connector (step 11) to the original federate (step 12). The results are similarly forwarded to the original federate (steps 13, 14, 15 and 16) The Federation Connector consults the policy how to handle the query results. The original federate combines the remote results with the local results (steps 17, and 18) and reports them to the client (step 19 and 20).

Interoperable InfoObject

Each implementation of the Infosphere can implement CAPI *InfoObject* differently. Thus we have designed InfoFedInfoObject to be a common denominator between incompatible Infosphere InfoObjects. It is used when objects are published and in query results are passed between federate with incompatible representation. The usage of *InfoFedInfoObject* introduces overhead for federation communication because of the need for double translation. In order to avoid unnecessary translation federates, when connecting, exchange namespaces of the InfoObject implementation. The *InfoFedInfoObject* is employed in the federation between two federates only when the classes differ.

Generalization of Federation Integration with Infospheres

Our experience in integrating with Apollo and Mercury has helped us determine what extension elements would be needed for other implementations. These *integration components* have to be introduced into the original CAPI-based infospace implementation in order to integrate Federation Service.

- *Subscription Receptor* is a component responsible for local publication of remote InfoObject. It receives the object from Federation Service through the call to its *newObject* method. If the subscription id matching this object is provided, the federate services are being used directly. If no subscription id is returned, it uses the local broker to locate subscribers.
- *Query Receptor* is responsible for receiving remote queries, processing them locally, and sending the query results back to the remote federate. It implements *startQuery* and *terminateQuery* methods. It calls local *RepositoryService* to execute the query on the local Archive.
- *Predicate Matcher* is used by the federation service to perform subscription matching on remote subscriptions. It needs to implement the *evaluate* method. It has to deal with the federate-specific representation of the predicate.
- *InfoObject Mapper* is responsible for constructing local representation of the *InfoObject* based on the *InfoFedInfoObject* received from the remote federate. It needs to fill up specific federate fields of the extended *InfoObject* with application data.
- *Policy Service Container* encapsulates the generic Federation Policy Service implementation. It makes the policy service available to the Federation Service. It needs to configure Policy Service using the local federate environment.
- *Federation Service Container* creates all the federation integration components and initializes them in the local infosphere. It also creates the generic Federation Service and Configures access to the federation properties needed by the Federation Service.

The actual bulk of the federation functionality has been developed as a generic functionality which can be integrated with any CAPI based infospace by developing the above listed integration components. The two subsequent sections will describe how theses integration components have been develop for Apollo and Mercury.

3.1.2 Approach for Apollo

The Federation Service for Apollo (Figure 7) is packaged as another MBean and has an associated helper class. It imitates other Apollo services which also are implemented as MBeans. It implements the generic Federation Service interfaces. It uses native Apollo Services to publish MIOs, get subscriptions and submit queries. The modified Apollo classes calling the Federation Service MBean are deployed as a patch to the original Apollo source code base. They implement the integration components.

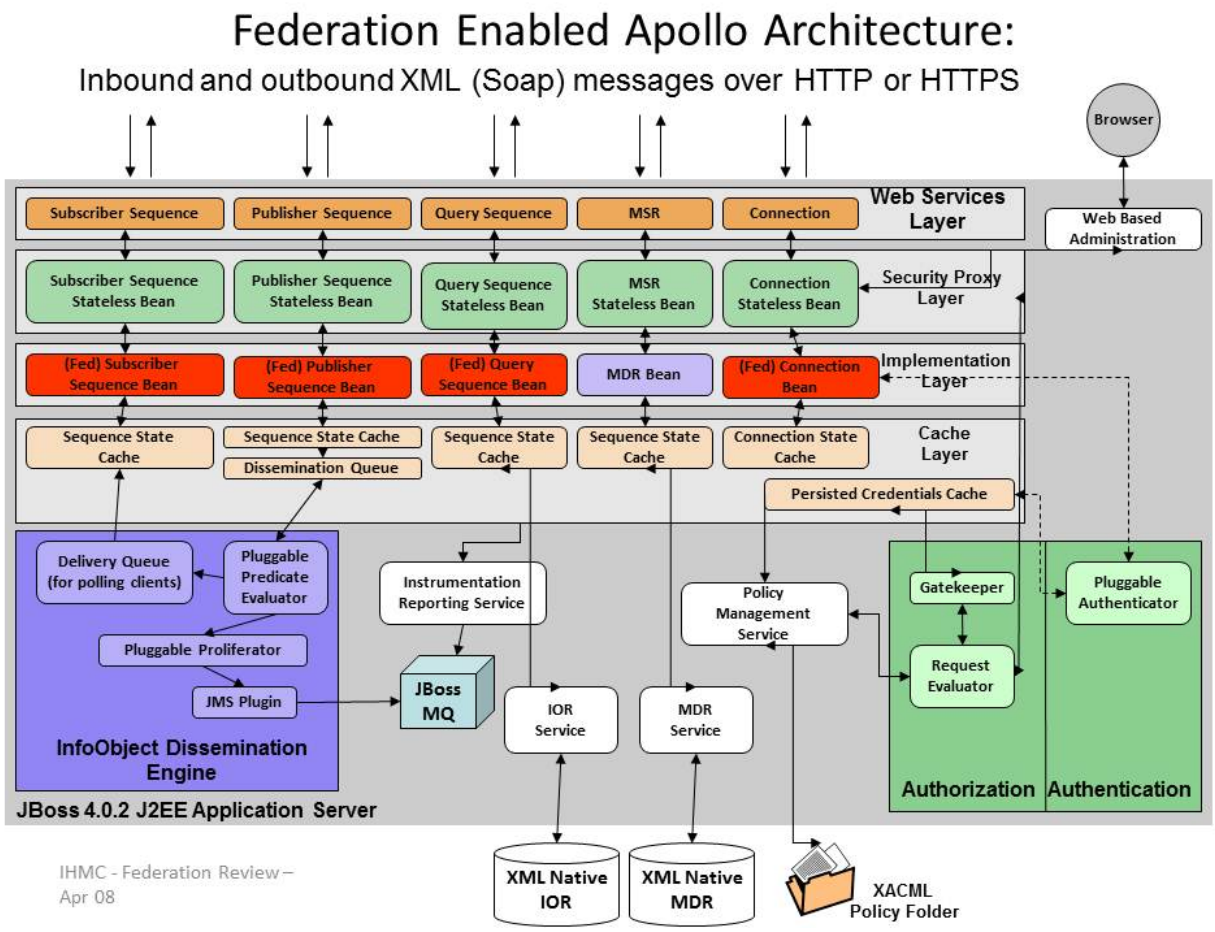

Figure 7: Federation Service components inside the Apollo Architecture

3.1.3 Initial Approach for Mercury

To develop the integration components the *MercuryFederationService* Component was created. It is installable component in OpenWings used by Mercury. It exports the generic FederationService interface. It defines *MercuryDisseminator* interface – implemented by *BrConnectionManager*. The integration also includes MercurySubscriptionReceptor to receive and disseminate remote MIOs. The Synchronous (RMI) Connector has been generated for *MercuryFederationService* Interface. *InstallableComponentDescriptorPolicy* has been created as well as an necessary Java Security Policy. Finally *BrConnectionManager*, *BrSubscribeSequenceManager*, *BrPublishSequenceManager* and *InstallableComponentDescriptorPolicy* have been modified. The Federation Query has not been integrated with Mercury. Additionally we developed GUI to show Federation-related Status Information for Mercury.

3.1.4 Evolving Approach for Phoenix

Following the services-based approach of Phoenix, the federation capability is realized through a set of services that work in conjunction with each other and the original Phoenix services. Four possible approaches to the integration have been designed and prototyped:

- Wrapping Approach; it has a single, integrated, Federation Service Component which appears as a single service to Phoenix. It ensures that all aspects of Federation are transparent to Phoenix. The Federation Service Component internals is the CAPI

originated federation services with translation mechanism to Phoenix specific datatypes. The component implements federation discovery, contracts, policies and adaptation mechanisms.

- The Phoenix Services Approach extends necessary Phoenix services to support Federation. The original Phoenix services modified include Broker, Dissemination Service, Query Service, and Discovery Service. It provides two versions of each Phoenix service. The original version and the federation-capable version. The run-time configuration allows switching between versions.
- The Service Composition Approach which dynamically instantiates and injects federation services into Phoenix. This process is managed via Service Orchestration and Workflows (e.g., BPEL)
- The Multiple (Native) Phoenix Services Approach; federation capabilities are realized as a set of Phoenix-level Services (Figure 8). This is the final approach. The architecture of this approach best matches the Phoenix services architecture. It allows selectively creating multiple instances of chosen federation services either for load-balancing or redundancy purposes. In this approach, the native Phoenix channel mechanism has also been adopted as the primary communication channel between federates, replacing direct TCP and Mocket connections.

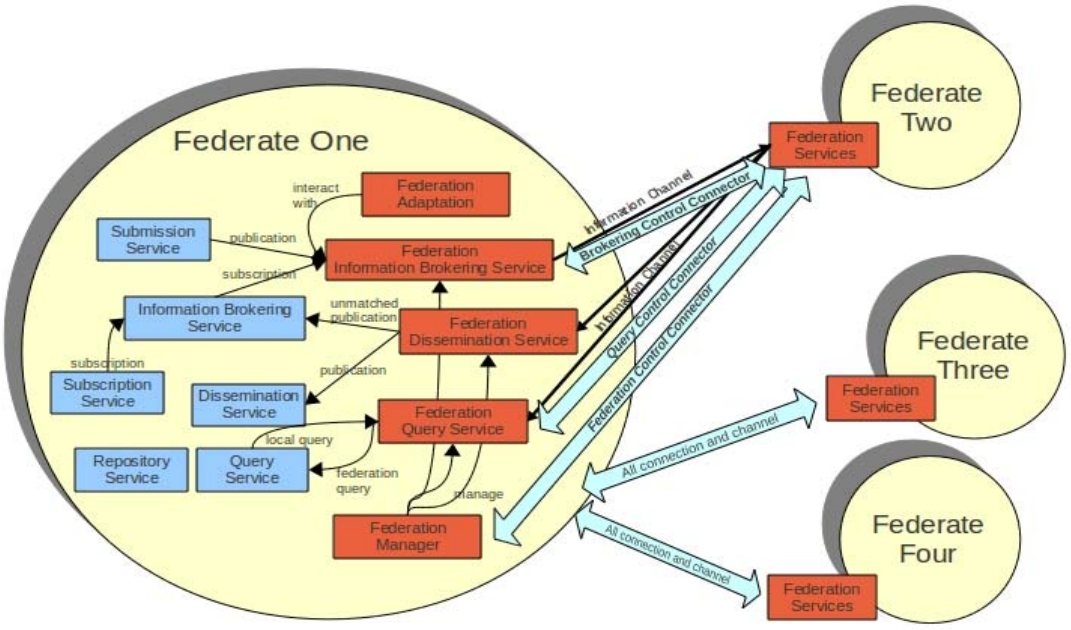

Figure 8: Final architecture of the Federation Service integration into Phoenix

Federation Manager Service

Once potential new federates are identified by the Discovery Manager, the Federation Manager (FM) Service is responsible for setting up the federation across the newly discovered entities. In particular the FM communicates with the new federates, negotiates contracts and informs the

other federation services about the new federates. The FM is also responsible for handling disconnections and termination of federation.

Federation Information Broker
Information brokering is one of the fundamental services performed by Phoenix. Brokering involves examining new, incoming information that has been published and matching it against active subscriptions from clients. Any matching information is then forwarded to the appropriate clients through the Dissemination Service. The Federation Information Brokering Service (FIBS) extends information brokering to handle federates. It receives subscription registrations from the Subscription Service and forwards them to the federates. It also receives the local publications from the Submission Service, brokers them locally on the behalf of the remote federates, and forwards them to appropriate federates. In particular, it forwards them to remote Federation Dissemination services (see below).

Federation Dissemination Service
Dissemination is the post-processing step that follows brokering and involves transmitting matched information to the clients. The Dissemination Service normally receives matched data from the Information Brokering Service. When federation is involved, the Federation Dissemination Service (FDS) is responsible for receiving matched information from remote federates that is destined to local clients. In most cases, when the FDS receives forwarded publications from remote federates, they have already been matched for the local clients (by the remote Federation Information Broker). In such cases, it uses local Dissemination Service to transmit the data to the clients. Otherwise, it uses the local Information Broker to publish the information locally.

Federation Query Service
Querying for archived information compliments publish and subscribe as the third core operation provided by Phoenix in the context of information management. Query differs from subscribe in being able to retrieve previously published and stored data. The query service permits information retrieval from the client's data stores and supports synchronous and asynchronous query execution. Data stores are managed by the Repository Service and they could be of two different kinds: repositories and archives. Repositories are low-latency high-access data stores that should support higher data read and write rates. Archives are expected to store much more data than repositories, but with a lower data access rate.

The Federation Query Service (FQS) extends the query capability to remote federates. It receives local queries and sends them for processing from both the remote federates and the local Query Service, collects the results, and returns them to the client. One of the assumptions made by the FQS is that federates do not have duplicated data, which simplifies the distributed query problem. The FQS may locally cache data that results from a remote query, thereby improving performance for repeated queries. The nature of the queries, as well as the behavior of the FQS, can be controlled via policy. For example, a query by a coalition partner being executed against a US database may be modified in order to limit the scope and nature of the query. This control is independent from the ability to control the individual objects that are a result of the query.

3.2 Discovery

Discovery refers to the process through which a node becomes aware of other nodes and the services they provide. Nodes make use of the discovery mechanism to register services and advertise the availability of computational resources through the dissemination of messages across the network.

Because of the lack of a fixed infrastructure and the presence of non-fixed nodes that characterize tactical networks, discovery is often accomplished by the broadcasting of packets. The most simplistic form of broadcasting, called *flooding*, typically causes unproductive and harmful bandwidth congestion as each node retransmits each received packet exactly once.

The initial approach used by Federation relied on the Group Manager, a component of the Agile Computing Middleware that supports peer-to-peer node and resource discovery [4] [5]. Subsequently, discovery was supported by the XLayer Adaptive Discovery and Group Service.

The approach that the XLayer offers is a hybrid discovery mechanism that is capable of self-adapting to different network topologies and traffic scenarios. This self-adapting mechanism monitors the network at different levels to make use of a broadcasting algorithm that is more suitable for a set of localized network conditions. The main goal of this mechanism is to reduce the number of retransmissions for broadcast packets while attempting to maintain the same delivery rate of more simplistic broadcast techniques that are known to cause undesired bandwidth congestion.

This adaptive mechanism makes use of different broadcasting algorithms that are known to perform better under certain network conditions. Based on the characteristics of the current network topology (i.e. sparse versus dense), the dissemination service would activate a broadcast algorithm that is more suitable for the given network topology and traffic conditions, reducing the number of retransmissions, and improving the effectiveness of the broadcasting algorithm and the overall performance of the dissemination mechanism.

Additionally XLayer offers a Grouping Service. By taking advantage of the adaptive dissemination described above nodes can permanently advertise group membership and perform peer searches within a certain scope.

The adaptive discovery using efficient broadcasting offered by XLayer is now being redesigned and reimplemented within VIA (Virtual Interface Approach to Cross-Layer Communications). VIA is a replacement implementation of XLayer that enables applications to better adapt and leverage the characteristics of the dynamic communication environment, and it also enables the underlying communications infrastructure to better support application requirements and constraints. VIA operates at the data link level: it creates a virtual network interface that applications can use to manage traffic over multiple physical network interfaces.

Discovery, Grouping and Federation

During the Federation establishment phase, the Discovery Manager component that runs within the Federation core set of services registers with the underlying XLayer for discovery and

grouping, through the Java XLayer Proxy. Then a predefined group for federation is created. Any IMS that joins such group manifests its desire of becoming part of the Federation.

The discovery controller in XLayer notifies federates about the presence of other potential federates, providing an endpoint –typically an IP and a port- where to reach each of them. This notification triggers a handshake phase between federates. Once this phase is completed the Federation is officially established.

If one of the nodes involved in the federation dies or leaves the federation group the XLayer notifies the other peers almost instantly. This way when a federate is disconnected or leaves the network all the references to it can be cleaned up by the other federates so that they will no longer attempt to send information it.

3.3 Monitoring

The Monitoring Service [6] was initially developed for Apollo, and was more recently migrated to the Phoenix architecture. It was designed to provide monitoring capabilities for the Federation environment, for the QoS-Enabled Dissemination set of services and in general, for any service within the architecture that may want to take advantage of it.

The Monitoring Service operates as a high-level interface for the monitoring functionalities of storing time-series containing sets of values for each desired metric and providing real-time statistics exposed by the XLayer [7] substrate.

The monitoring API includes two different sets of functionalities: the first set manages the registration of new metrics provided by monitoring components and enables the update of existing metrics. The second set of operations allows other services to retrieve statistics about the currently monitored metrics, either by polling or via the subscription mechanism.

The Monitoring Service takes care of interacting with the XLayer substrate that supports the metric storage and retrieval at a lower level. In addition XLayer incorporates a set of built-in system related metrics (e.g., CPU and memory utilization and network traffic per interface). Figure 9 shows the architecture of the Monitoring Service highlighting how it is placed into the IM services.

The next paragraphs describe the Monitoring Service in more detail as a service in the context of Phoenix. The functionalities that were initially developed for Apollo are a subset of the ones presented below. Due to the distributed nature of Phoenix we had to extend and improve the monitoring architecture in order to best fit the Phoenix environment.

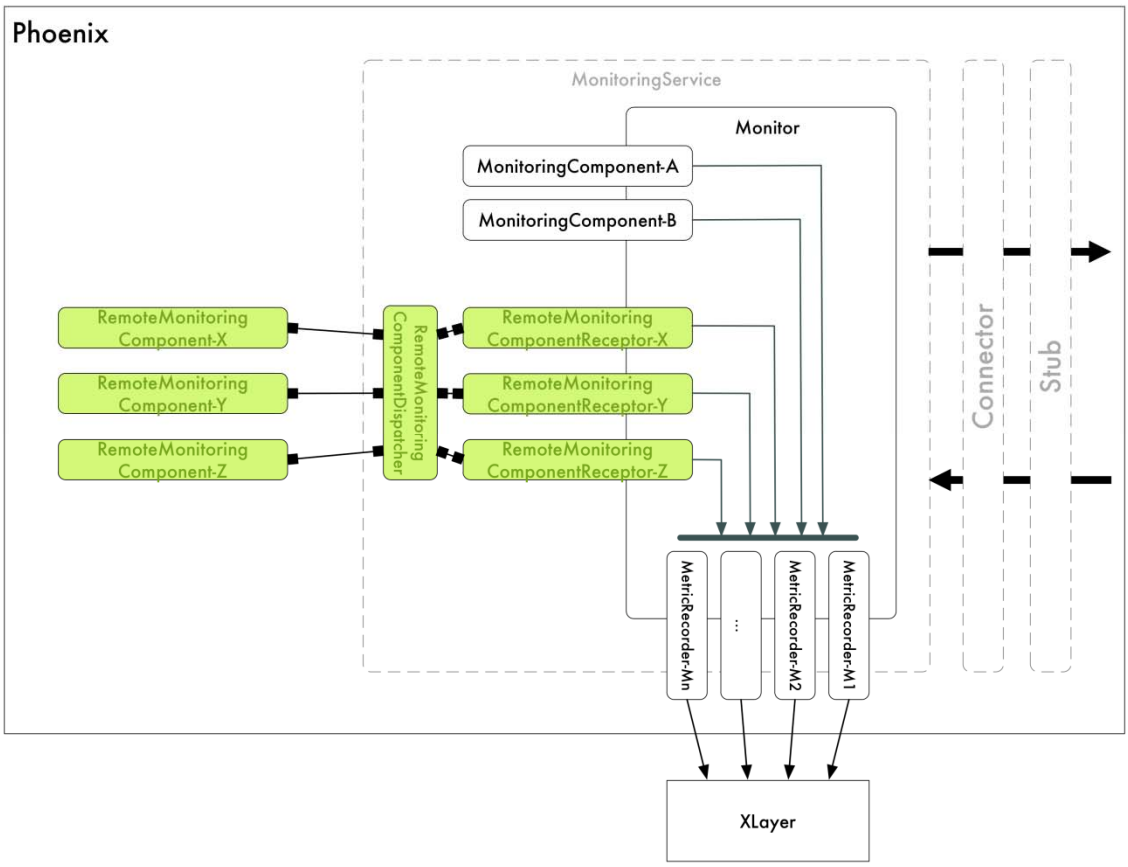

Figure 9: Monitoring Service for Phoenix

Monitoring Service for Phoenix

The Monitoring Service for Phoenix provides monitoring capabilities to the entire architecture. In addition to the set of services that compose the Federation architecture, other services within Phoenix may take advantage of this service to store and retrieve values for metrics that need to be monitored.

Collecting metric information: Monitoring Components

One of the key aspects of the Monitoring Service is its extensibility and flexibility. Other services that are part of the Information Management System may define their own Monitoring Components that can be dynamically plugged-in to the main Monitoring Service. Monitoring Components define customized metrics they wish to monitor, registering them with the Monitoring Service and specifying themselves as providers. Once a metric is registered the Monitoring Service returns (and stores) a reference to the related Metric Recorder. Metric Recorders expose the API to update the values for the metrics to which they are related.

Collecting metric information: Remote Monitoring Components for Phoenix

Given the distributed nature of Phoenix, with services potentially sitting on different nodes, the Monitoring Components that are plugged in the Monitoring Service must provide remote monitoring capabilities. For this reason, in addition to the regular Monitoring Components we developed Remote Monitoring Components. Their API enables dynamic remote registration, deregistration and update of metrics in a completely transparent way to services that wish to use

monitoring functionalities. Upon registration of a Remote Monitoring Component with the central Monitoring Service via the corresponding stub, a related Remote Monitoring Component Receptor is created. The role of the Receptor is to handle the interaction with the Metric Recorders mentioned above while receiving metric information and control messages from the associated remote component. The communication between the Remote Monitoring Components can be done using any protocol. For simplicity the default implementation uses TCP, but it would be easy to switch to Mockets or even to Phoenix's channels.

Accessing metric statistics

The statistics collected and aggregated by the Monitoring Service can be accessed either by polling or using a subscription mechanism, both exposed by the service (and service stub) interface. The polling API supports the listing of all the available metrics at the time of the invocation and for each of the metrics the retrieval of the aggregated statistics, i.e. last value, average value, variance and trend. The subscription API (currently working both locally as well as remotely via RMI) provides two different ways of subscribing. Persistent subscriptions enable subscribers to be notified every time metrics they are interested in are updated. On the other hand, one-time subscriptions with threshold are triggered the first time the related metrics are updated exceeding the predefined range of values.

Configurable Monitoring Component and Phoenix events

To better integrate the Monitoring Service inside the Phoenix architecture and to fully take advantage of the capabilities offered by the framework, the Monitoring Service was extended to include event notification capabilities.

Specifically, the latest version of the Monitoring Service can be configured to monitor a predefined set of metrics of interest. When the service starts up, a properties file is read to extract the configuration parameters for such metrics and the needed Metric Recorders are instantiated. At the same time a Phoenix Subscription Proxy is created: its role is acting as a proxy between the native subscription support provided by the Monitoring Service and the Event Notification Service available in Phoenix. In particular, the Phoenix Subscription Proxy issues a set of subscriptions with thresholds and registers itself as the recipient for the callbacks for such subscriptions. Every time one of the observed metrics falls outside the specified range the proxy is notified. Notifications cause the triggering of a Metric Updated Event via the Event Notification Service; hence any service interested in such events will be eventually notified. Figure 10 highlights the components that support these capabilities.

Some metrics, because of their nature, may not be known at configuration time. Client-related metrics are a good example: they only live as long as the corresponding client is alive, i.e. connected to Phoenix. In order to support this, the Configurable Monitoring Component as well as the Phoenix Subscription Proxy allows for wildcards in the metric configuration file. Therefore, the Phoenix Subscription Proxy will be notified of relevant metrics regardless of the specific client identifications.

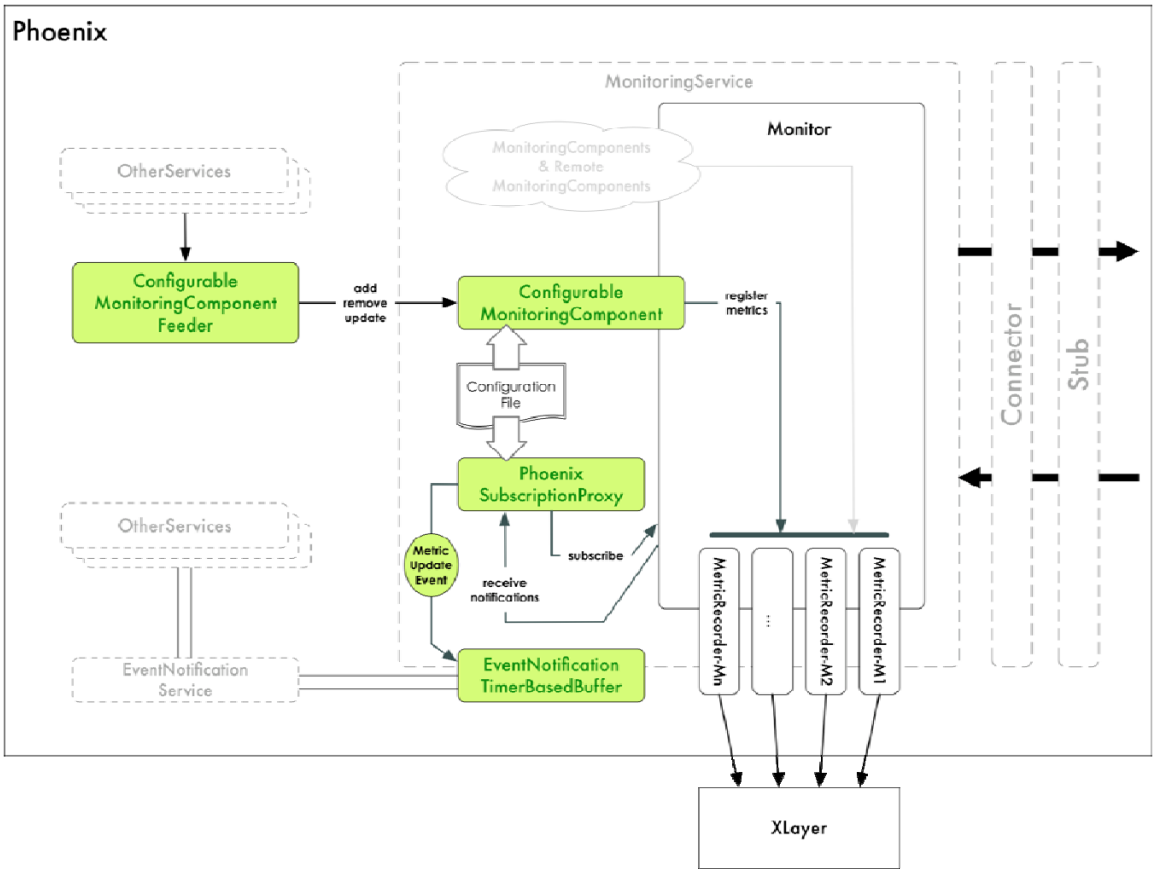

Figure 10: Updating and Notification of Metrics in the Monitoring Service

Monitoring Service and Federation

As any other service that wishes to use the monitoring functionalities, the Federation code implements its own Federation Monitoring Component that exposes all the functionalities for monitoring the behavior of the services that are part of Federation. The Federation Monitoring Component registers with the underlying Monitoring Service as a provider for application-level statistics about the performance of the local IMS (e.g., number of info objects published per second, predicate matching rate per subscription, etc.). The Adaptation Manager can then take advantage of the application-level statistics along with information about system and network behavior to dynamically adapt the behavior of federation.

3.4 Policies and Contracts

The key to coordinated operation of federated infospheres is a comprehensive, semantically-rich, and enforceable service agreement. The privileges and obligations of each infosphere within the federation must be established and monitored for compliance at all times. The service agreement binds all parties to act according to the constraints accepted when the federation was formed. This approach is necessary to ensure the proper flow of information through the federation. The KAoS based Federation Policy and Contract Service is used by the Federation Service to create and enforce federation contracts.

This service has been integrated with the Federation Service for Apollo. It has not yet been ported to Phoenix Federation Services. The primary reason for this was that Phoenix was still

undergoing development during the course of the Federation project effort. Therefore, given the timeframe allowed, our effort was focused on developing and updating basic federation services to the new version of Phoenix, and did not allow us to complete porting the policy and contract mechanism.

3.4.1 Technical Overview of the KAoS Services Framework

KAoS, a set of platform-independent services, enables people to define policies ensuring adequate security, configuration, predictability, and controllability of distributed systems, including traditional distributed platforms (e.g., CORBA, Web Services, Grid Services), software agent frameworks (e.g., NOMADS, Cougaar), and multi-robot configurations. KAoS Domain Services provide the capability for groups of software components, people, resources, roles, groups, and other entities to be semantically described and structured into organizations of domains and subdomains to facilitate collaboration and external policy administration. KAoS Policy Services allow for the specification, management, conflict resolution, and enforcement of policies within domains. KAoS policies distinguish between authorizations (i.e., constraints that permit or forbid some action by an actor or group of actors in some context) and obligations (i.e., constraints that require some action to be performed when a state- or event-based trigger occurs, or else serve to waive such a requirement).

Policies are represented in ontologies, not rules. The use of ontologies, encoded in OWL (Web Ontology Language, http://www.w3.org/TR/owl-features/), to represent policies enables reasoning about the controlled environment, about policy relations and disclosure, policy conflict resolution, as well as about domain structure and concepts. KAoS reasoning methods exploit description-logic-based subsumption and instance classification algorithms and, if necessary, controlled extensions to description logic (e.g., role-value maps).

KAoS Architecture. Two important requirements for the KAoS architecture have been modularity and extensibility. These requirements are supported through a framework with well-defined interfaces that can be extended, if necessary, with the components required to support application-specific policies. The basic elements of the KAoS architecture are shown in Figure 11; its three layers of functionality correspond to three different policy representations:

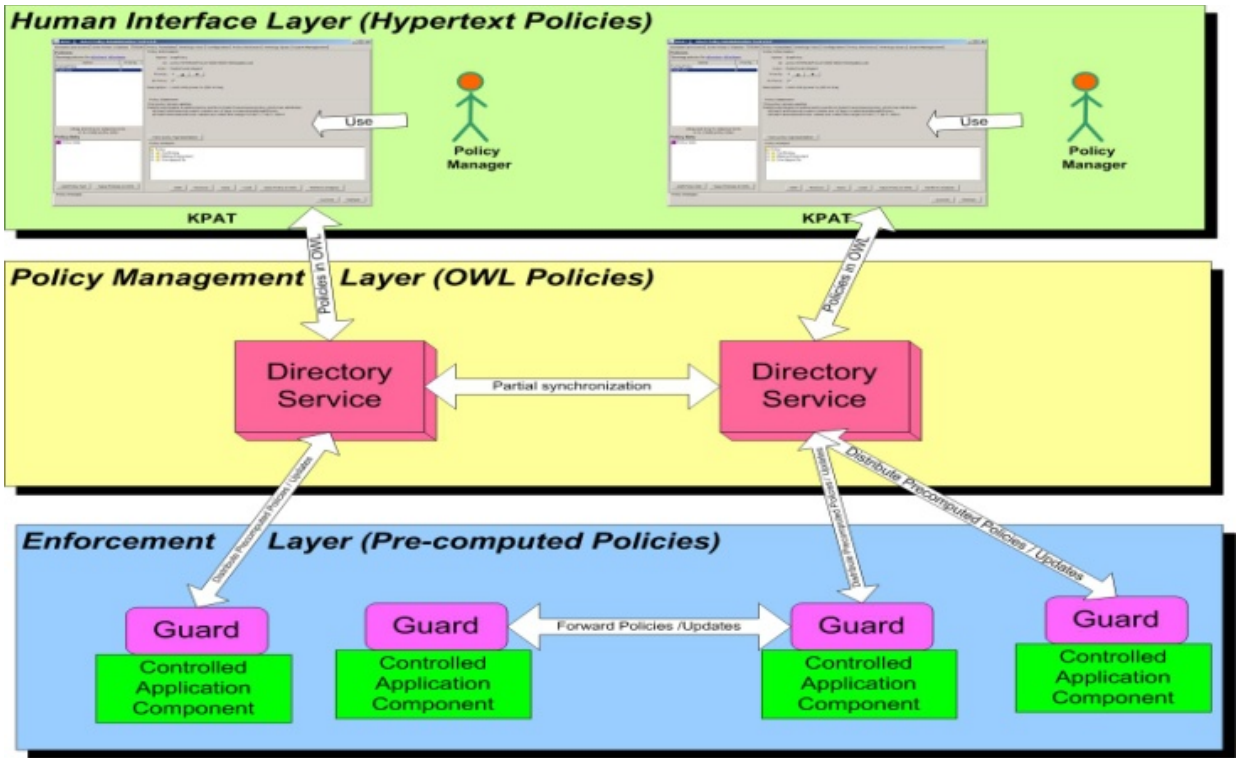

Figure 11: KAoS Policy Services Conceptual Architecture

Human Interface layer: This layer uses a hypertext-like graphical interface for policy specification in the form of natural English sentences. This capability, called KPAT (KAoS Policy Administration Tool), hides the complexity of OWL from users, and provides the ability to analyze, monitor, and manage ontologies and policies. Further simplification of the policy specification task is possible through Policy Templates and Wizards. The vocabulary for policies is automatically provided from the relevant ontologies, consisting of highly-reusable core concepts augmented by application-specific ones. Unlike most other policy frameworks, changes of any kind can be made efficiently at runtime.

Policy Management layer: Within this layer, OWL is used to encode and manage policy-related information. The Distributed Directory Service (DDS) encapsulates a set of OWL reasoning mechanisms.

Policy Monitoring and Enforcement layer: KAoS automatically “compiles” OWL policies to an efficient format that can be used for monitoring and enforcement. This representation provides the grounding for abstract ontology terms, connecting them to the instances in the runtime environment and to other policy-related information.

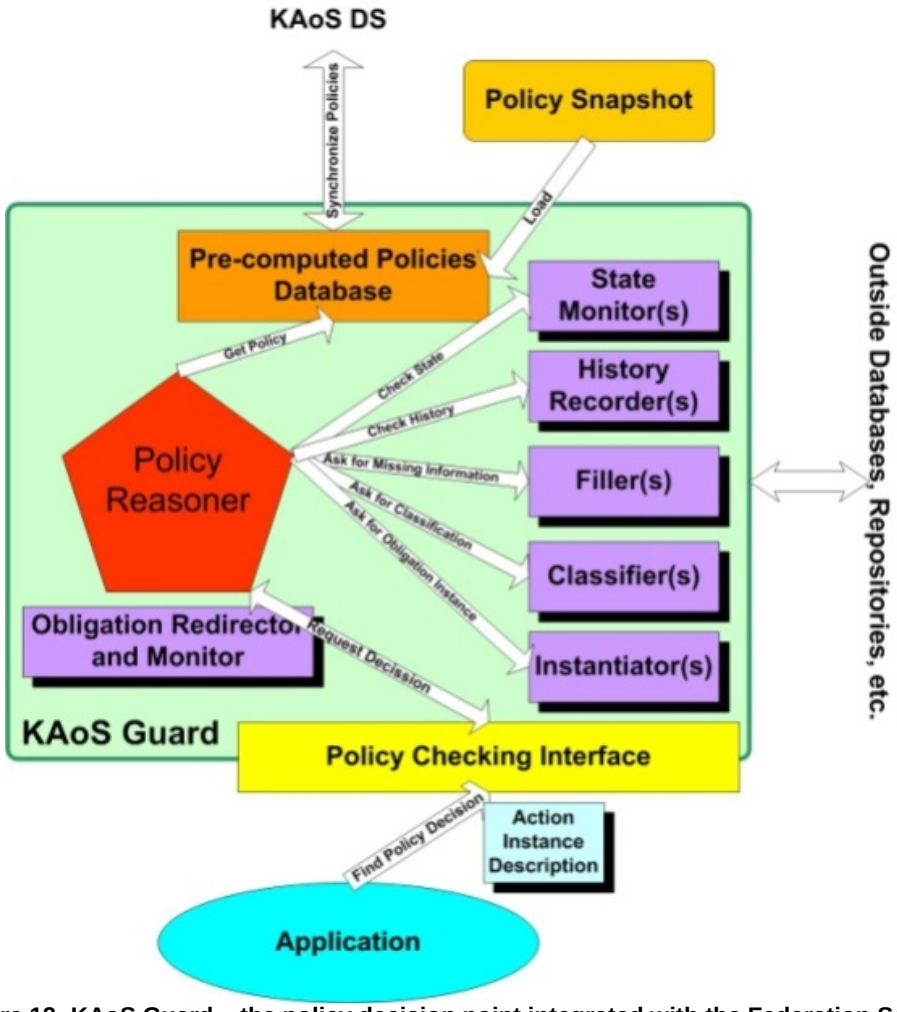

Figure 12: KAoS Guard – the policy decision point integrated with the Federation Service

3.4.2 Controlling Federation

The FS on each federate is integrated with the KAoS Guard software component (Figure 12), which stores policies controlling establishment, lifecycle, information exchange and adaptation of the federations established by this federate.

The Guard interface has been customize for the federation functions with methods allowing checking authorization of different federation actions.

```
public interface FederationPolicyService
{
    public void registerFederate (String federateName, String federationName);
   public void registerRemoteFederate (String federateName);

   public void deregisterRemoteFederate (String federateName);
   public void setFederateProperty (String federateName, String propertyName, String properyValue,
         String valueDescription);
   public boolean authorizeNewFederateConnection (String remoteFederateName);

   //--------------------------------
```

```
    // Methods dealing with authorization and modification of federation subscriptions
    //---------------------------------
    public Object authorizeForwardingOfSubscription (String remoteFederateName, Object infoObjectType,
            String infoObjectTypeVersion, Object subscriptionPredicate);
    public Object authorizeAcceptingOfRemoteSubscription (String remoteFederateName, Object infoObjectType,
            String infoObjectTypeVersion, Object subscriptionPredicate);

    //---------------------------------
    // Methods dealing with authorization and modification of federation publishing
    //---------------------------------
    public Object authorizeForwardingOfInfoObject (String remoteFederateName, Object infoObjectType,
            String infoObjectTypeVersion, Object publishedEntity);
    public Object authorizeAcceptingOfRemoteInfoObject (String remoteFederateName, Object infoObjectType,
            String infoObjectTypeVersion, Object publishedEntity);

    //---------------------------------
    // Methods dealing with authorization and modification of federation query
    //---------------------------------
    public Object authorizeForwardingOfQuery (String remoteFederateName, Object infoObjectType,
            String infoObjectTypeVersion, Object queryPredicate);
    public Object authorizeAcceptingOfRemoteQuery (String remoteFederateName, Object infoObjectType,
            String infoObjectTypeVersion, Object queryPredicate);
}
```

This interface has been simplified; instead of separate methods authorizing the actions and informing about the required changes, we implemented a single method for each action control returning:

- Null – if the action is not authorized,
- Modified Predicate (for subscription and query) or Metadata (for publication and query results),
- Original Predicate or Metadata if no changes are required.

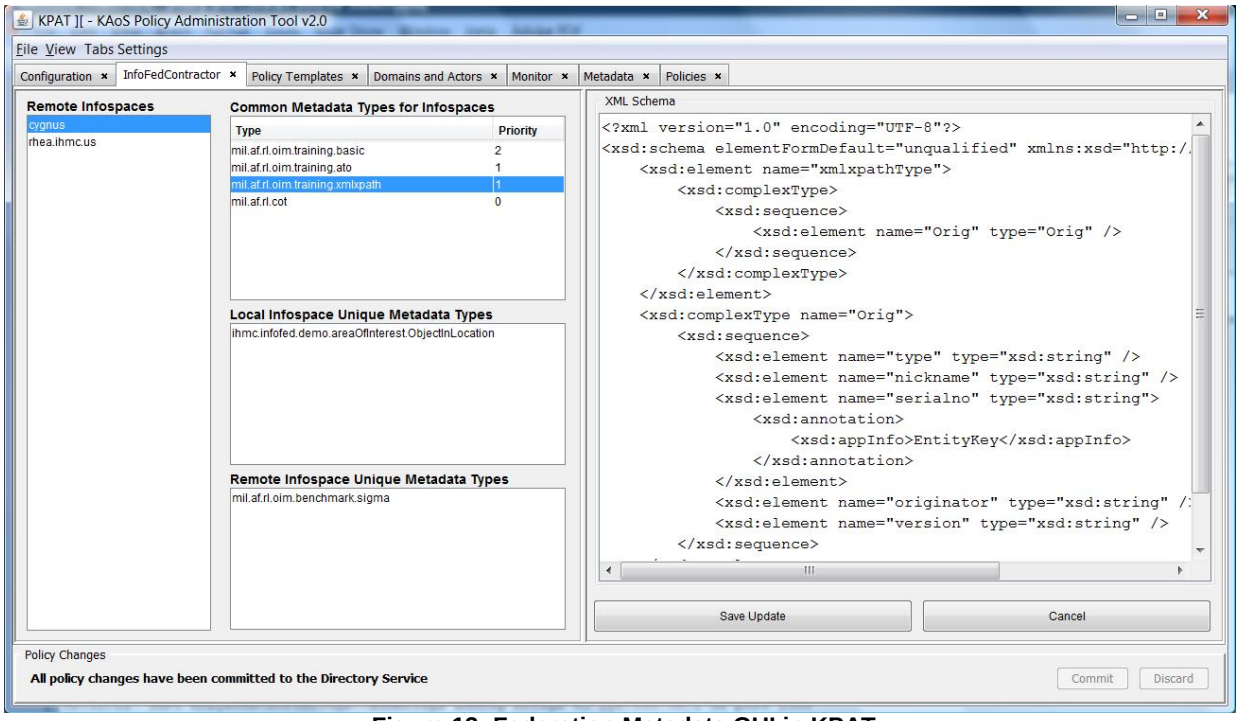

Figure 13: Federation Metadata GUI in KPAT

KPAT draws information from the local and remote federate metadata repositories and allows the user to examine local and remote federate metadata types and their schemas and to define priorities specific to metadata types for the given remote federate.

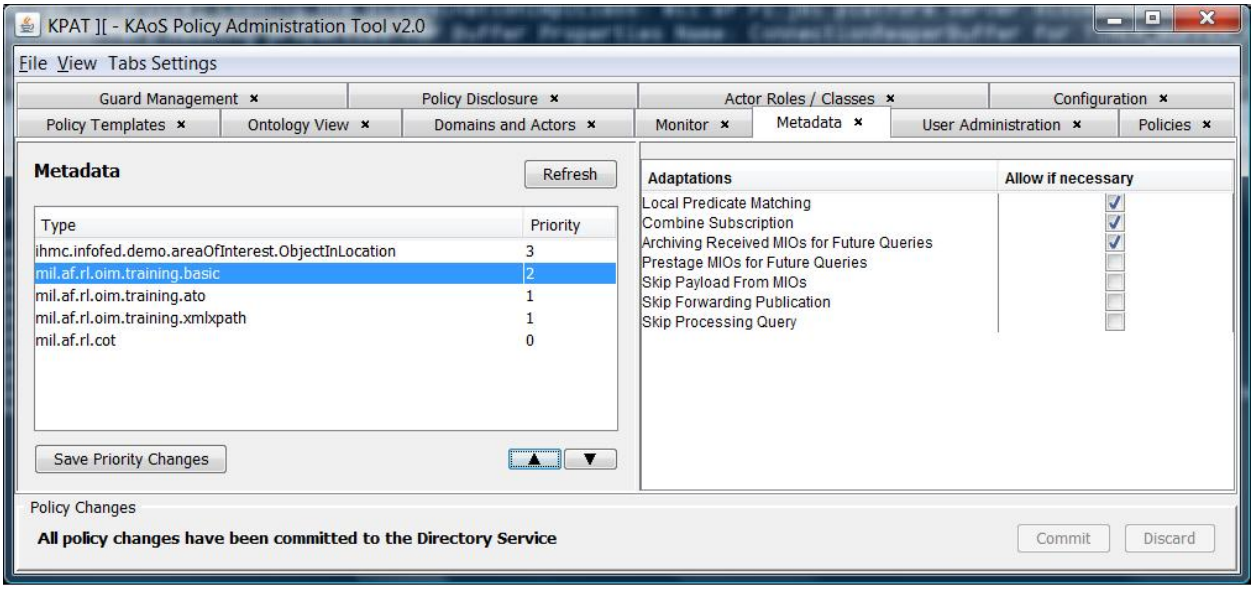

Figure 14: Contract GUI in KPAT

The Contract GUI in KPAT allows the user to prioritize metadata types and to select which adaptation strategies are acceptable by this federate. This default contract setting is sent to the remote federate. It contains:

- A List of metadata types it potentially intends to subscribe to or query about with importance priority attached to it
    - The metadata types have priorities associated with itself depicting importance of the information of this type to the federate clients
    - This list displays the metadata types the local clients will be able to use
- An Adaptation matrix
    - Informing what types of adaption can be used or should be used on the connection with this federate
- The type of the Information Object used by the federate
    - Will determine if the usage of the *Interoperable InfoObject* would be necessary

When a node discovers a new possible federation partner and the initial connection is established, the two potential federates exchange information about their current configuration. Based on this information as well as its own local policies, each federate independently decides:

- Whether to establish a federation with the remote federate,
- What priority to attach to the remote federate,
- Based on the current resource usage for the federation operations and the assigned federate priority, how to estimate the quantity of resources it can devote to server requests from the federate,
- What metadata type subscriptions or queries it would be able to support for a given federate.

As a next step, based on the publication level and archive size for the given metadata type (obtained from monitor), estimate for each requested metadata type in the contract the expected utilization of local resources for processing publication and queries.

The metadata types are processed according to their priorities from the contracts. When a certain limit of resource usage is reached either the rest of metadata types will be not supported or certain adaptation will be employed for them. It can result in a Contract Revision, which is sent to the remote federation with information about what metadata types will be supported and in what fashion.

Policies controlling this contract revision process can forbid sending information about certain metadata types based on the properties of the remote federate. Further, the policies can limit the usages of local resources to support the federate based on the local priority assigned to the federate. Policies can also trigger usage of certain adaptation after reaching a certain limitation of calculated resource usage based on local priority assigned to the federate and the adaptation matrix from the Contract Proposal.

The remote federate analyzes the contract revision. It accepts the revision if its minimal request was accepted and if its local expected resource usage by the remote federate is balanced with promised services. As a result, it either accepts the revision and continues with the federation or rejects the revision and terminates the federation. The counter offer mechanism has not been considered.

Policies setting characteristics for acceptable contract revision are base on the property of the remote federate, the list of required supported metadata types, and the list of employed adaptations.

If the federation is established then during subsequent subscription exchanges, queries, and publication with federates, each operation is analyzed with respect to current policies. Policies may allow or prevent a given operation. They may also modify the operation by changing the subscription or query predicate, or by removing metadata from the published information object being forwarded to the remote federate. Moreover, they may enforce or waive obligations (e.g., logging) relevant to certain types of operations. In addition, policies and the agreed adaptation policies control how and when a given adaptation mechanism is activated when the share of resources used by the given federate exceeds the agreed-upon limit. KPAT configuration for the control of federation consists of sets of predefined policy templates and policies associated with them. Each policy can be easily activated and deactivated. The policy templates are grouped into four categories:

- Federation Acceptance Polices,
- Gatekeeping Policies,
- Adaptation Policies,
- Contract Policies.

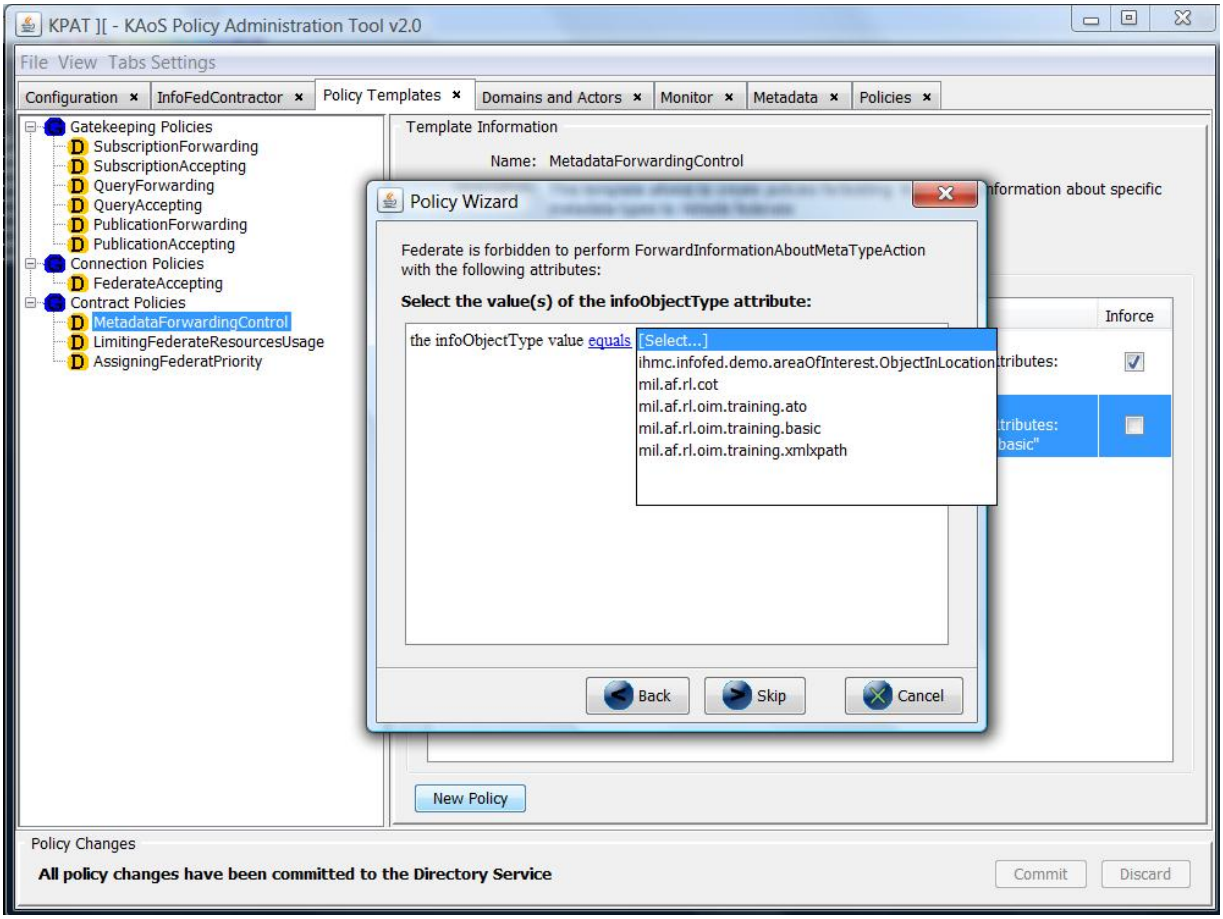

Figure 15: KPAT configuration for the Federation Service Policy

The GUI allows the management of federation policies using user friendly wizards to hide the complexity for the underlaying represention of policies.

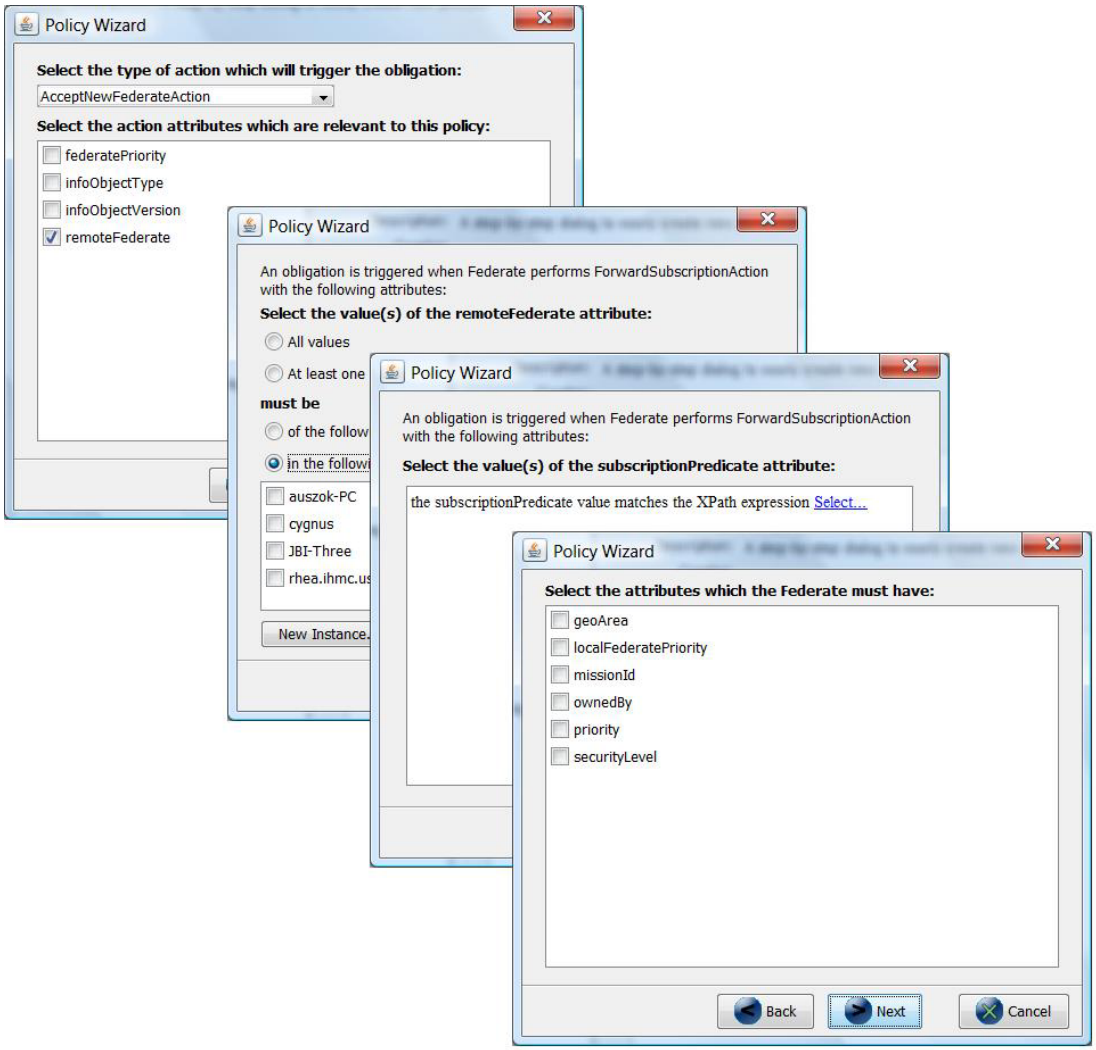

Figure 16: Policy Wizard for the definition of federation policies

3.4.3 Contract Negotation Example

The example presents federation between two federates – Alpha and Beta. Federate Alpha possesses a Predator UAV and thus produces the following metadata types with the following publishing rates:

- mil.af.predator.radar 1-2 Hz
- mil.af.predator.stillDayTime 1-2 Hz
- mil.af.predator.stillInfrared 1-2 Hz
- mil.blueforcetrack 1 Hz
- mil.spotreport 1/15 Hz

The federate already provides intelligence data to a few Air Force and US Army units connected as clients to this information space. Information processing currently consumes about 80% of the CPU and 75% of the network bandwidth (from the ground station).

Federate Beta is a NATO Country Army Unit, which attempts to create federation with Federate Alpha.

Contract/Policy Area	Contract Proposal Federate Beta	Policies Federate Alpha	Agreed Contract for Federate Beta
Requested Access to Subscription Data	mil.af.predator.radar mil.af.predator.stilllnfrared mil.blueforcetrack mil.spotreport	For Non USA Federate No access to mil.af.predator.still Max additional CPU usage 5%	mil.af.predator.radar mil.blueforcetrack 1/5 Hz mil.spotreport
Requested Access to Query Data	mil.af.predator.stilllnfrared mil.blueforcetrack mil.spotreport	Above policies plus: No query allowed for mil.spotreport	mil.af.predator.stilllnfrared max 1/120 Hz max result size 200 objects mil.blueforcetrack max 1/60 Hz
Predicate Matching Adaptation	No aggregated predicate	No policy	Will use remote predicate matching if necessary
Deferred payload handling for subscription	No for: mil.af.predator.radar	Require if needed	Will use it for all subscription if necessary
Query pre-staging	mil.blueforcetrack mil.spotreport mil.af.predator.stilllnfrared	Only allow pre-staging for reports	mil.blueforcetrack mil.spotreport
Query deferred payload	No for all	Require if needed for still	Will use it if necessary for mil.af.predator.stilllnfrared

Figure 17: Contract Example Details

Contracts are created on both sides of the federation but below we present contract creations for Federate Beta on the Federate Alpha side. The subsequent sections of the contract, relevant policies and the result of the modification are presented below.

Requested Access to Subscription Data
Federate Beta requested in its contract subscription access to the following types of information with the following priority:

- mil.af.predator.radar
- mil.af.predator.stillInfrared
- mil.blueforcetrack
- mil.spotreport

Federate Alpha's local policies forbid access to mil.af.predator.still by any non-US Federate. Additionally, the policies limited the amount of resources the federate uses for processing of information dedicated to the non-US federates to maximum CPU utilization of 5%.

The contract service analyzes the forecasted need of processing power (provided by the Federation Monitoring) and provides the following modifications allowing access to the information of type:

- mil.af.predator.radar

- mil.blueforcetrack 1/5 Hz (reduced frequency)
- mil.spotreport

Requested Access to Query Data
Federate Beta requested in its contract query access to the following types of information with the following priority:

- mil.af.predator.stillInfrared
- mil.blueforcetrack mil.spotreport
- mil.spotreport

Since query is a type of access policy forbidding access to mil.af.predator.still by any non-US Federate also applies here. In addition there a special policy restricting query on mil.spotreport. As a result the modified contract allows query for:

- mil.af.predator.stillInfrared but max 1/120 Hz max and result size not bigger than 200 objects
- mil.blueforcetrack but max 1/60 Hz

In addition Federate B specifies its preferences on adaptation mechanism such as: predicate matching adaptation, deferred payload handling for subscription, query pre-staging or query deferred payload. For instance, not authorizing deferred payload forwarding for information type: mil.af.predator.radar.

This contract will govern how data will be published and query performed from Federate Beta on Federate Alpha.

3.5 Adaptation

During the course of operations, the resources available for information management are likely to change over time. For example, the network links connecting federates may become saturated, or the systems hosting federation services may become overloaded. The Federation Adaptation Service performs local adaptations to offset such shortage of resources. For example, under low-bandwidth situations, the Adaptation Service can temporarily suspend low-priority subscriptions in order to provide reasonable performance for the remaining subscriptions. The priorities of the subscriptions can be specified via the client or via policies. On the other hand, when computational resources fall short, the Adaptation Service temporarily disables local predicate processing. This causes the Federation Information Broker to send all publications to the remote federate, and for the brokering to occur on the remote federate. Subscriptions are sorted based on their hit-rate (i.e., the percentage of publications that match the predicate) and the subscription with the highest hit-rate is selected first. This minimizes the impact of an increase in the bandwidth utilization as a result of this adaptation.

The Federation Adaptation Service was the last service integrated into Phoenix. While the architecture of the adaptation service is generic, we realized two specific behaviors to illustrate the operation: handling CPU overload on one of the federates and handling network overload over a link between two federates.

In particular, the adaptation service subscribes to the Monitoring Service for monitoring the CPU, and relies on the feedback information provided by the Mockets communications library for the status of the network links. As described above, a CPU overload is handled by turning off local predicate matching for remote subscriptions, and a network overload is handled by suspending low priority subscriptions.

One of the aspects addressed by the adaption service is the need for hysteresis, in order to prevent the service from continually adapting between good and bad states. As a simple example, consider the scenario where, with n subscriptions and k bandwidth, the channel becomes overloaded. The adaptation service may decide to disable one subscription. Now, the system observes a small increase in bandwidth, which might cause it to re-activate the disabled subscription, only to observe that the bandwidth is still not adequate, and the subscription must be disabled again. In order to prevent the adaptation service from vacillating, we have developed the notion of a state memory, which would allow the adaptation service to essentially learn good and bad states. This introduces hysteresis into the adaptation mechanism and reduces the number of times the adaption service would cause the system to re-enter a bad state.

The Federation Adaptation Service has two different control connections to services within the federate. The first connection is between the adaptation service and the Federation Information Brokering Service, which is utilized to suspend and resume subscriptions during the low-bandwidth adaptation mechanism. The second one involves the Event Notification Service and it is used to subscribe for Monitoring Service events for CPU overload.

A Phoenix byte channel is also established with the Federation Adaptation Service of each of the other federates that are currently a part of the federation. This allows the local adaptation service to notify the remote adaptation services about local adaptations and network information.

The following XML code is the portion of the spring configuration file which regards the Federation Adaptation Service.

```
<bean id="federationAdaptationService" class="us.ihmc.infofed.federationAdaptation.FederationAdaptationService">
 <constructor-arg>
  <bean class="us.ihmc.infofed.federationAdaptation.context.FederationAdaptationServiceContext">
   <property name="serviceName" value="MainFederationAdaptationService" />
   <property name="serviceTypes">
    <list>
     <value>FEDERATION_ADAPTATION</value>
    </list>
   </property>
   <property name="performeAdaptation" ref="PERFORME_ADAPT" />
   <property name="mocketMode" ref="MOCKET_MODE_FOR_ADAPTATION" />
   <property name="ipAddress" ref="MY_IP" />
   <property name="port" ref="FED_ADP_INPUT_PORT" />
  </bean>
 </constructor-arg>

 <property name="channelManager">
  <bean class="mil.af.rl.phoenix.service.channel.manager.BaseChannelManager">
   <constructor-arg>
    <bean class="mil.af.rl.phoenix.service.contexts.ChannelManagementContext">
     <property name="inputChannelMap">
      <map>
       <entry key="FederationAdaptation.InputChannel.1">
        <bean class="mil.af.rl.phoenix.channel.ChannelContext">
```

```
            <property name="channelType" value="BYTE" />
            <property name="name" value="FederationAdaptation.InputChannel.1" />
            <property name="applicationLevelContext">
              <bean class="mil.af.rl.phoenix.channel.data.ApplicationLevelContext">
                <property name="protocolId" value="simple" />
              </bean>
            </property>
            <property name="endPointContext">
              <bean class="mil.af.rl.phoenix.channel.EndPointContext">
                <property name="hostAddress" ref="MY_IP" />
                <property name="hostPort" ref="FED_ADP_INPUT_PORT" />
              </bean>
            </property>
            <property name="transportLevelContext">
              <bean class="mil.af.rl.phoenix.channel.transport.TransportLevelContext">
                <property name="protocolId" value="tcp" />
              </bean>
            </property>
          </bean>
        </entry>
      </map>
    </property>

    <property name="inputManagerMap">
      <map>
        <entry key="FederationAdaptation.InputMGR.1">
          <bean class="mil.af.rl.phoenix.service.managers.input.TimerBasedBufferByteCallbackInputManager">
            <constructor-arg>
              <bean class="us.ihmc.infofed.federationAdaptation.FederationAdaptationServiceWorker" />
            </constructor-arg>
            <constructor-arg value="MainFederationAdaptationService" />
            <constructor-arg value="FederationAdaptation.InputMGR.1" />
          </bean>
        </entry>
      </map>
    </property>

    <property name="channelToManagerMap">
      <map>
        <entry key="FederationAdaptation.InputMGR.1">
          <list>
            <value>FederationAdaptation.InputChannel.1</value>
          </list>
        </entry>
      </map>
    </property>
  </bean>
 </constructor-arg>
</bean>
</property>

<property name="controlChannelManager">
 <bean class="mil.af.rl.phoenix.service.channel.control.manager.BaseControlChannelManager">
  <constructor-arg>
   <bean class="mil.af.rl.phoenix.service.contexts.ControlChannelManagementContext" >
    <property name="stubMap">
     <map>
      <entry key="ENS-Stub-For-FED-ADP">
       <bean class="mil.af.rl.phoenix.eventnotification.control.stubs.RMIEventNotificationServiceStub">
        <property name="context">
         <bean class="mil.af.rl.phoenix.contexts.StubContext">
          <property name="connectorName" value="ens-connector" />
          <property name="connectorAddress" ref="RMI_SERVER_ADDRESS" />
          <property name="connectorPort" ref="RMI_PORT" />
         </bean>
        </property>
       </bean>
      </entry>
      <entry key="FED-IBS-Stub-For-FED-ADP">
```

```
          <bean class="us.ihmc.infofed.federationinformationbrokering.control.stubs.RMIFederationInformationBrokerServiceStub">
            <property name="context">
              <bean class="mil.af.rl.phoenix.contexts.StubContext">
                <property name="connectorName" value="federationBrokerConnector" />
                <property name="connectorAddress" ref="RMI_SERVER_ADDRESS" />
                <property name="connectorPort" ref="RMI_PORT" />
              </bean>
            </property>
          </bean>
        </entry>
      </map>
    </property>
  </bean>
 </constructor-arg>
</bean>
</property>

<property name="serviceMultiplexor">
 <bean class="mil.af.rl.phoenix.service.multiplexor.DefaultServiceMultiplexor">
  <constructor-arg>
   <bean class="mil.af.rl.phoenix.service.contexts.ServiceMultiplexorContext">
    <property name="conditionMap">
     <map>
      <entry key="mil.af.rl.phoenix.subscription.SubscriptionContextInterface">
       <list>
        <value>FED-IBS-Stub-For-FED-ADP</value>
       </list>
      </entry>
      <entry key="mil.af.rl.phoenix.event.EventInterface">
       <list>
        <value>ENS-Stub-For-FED-ADP</value>
       </list>
      </entry>
     </map>
    </property>
    <property name="dispatchContainerMap">
     <map>
      <entry key="FED-IBS-Stub-For-FED-ADP">
       <bean class="mil.af.rl.phoenix.service.contexts.DispatchContainerContext">
        <property name="returnType" value="STUB" />
        <property name="nextEntityId" value="FED-IBS-Stub-For-FED-ADP" />
       </bean>
      </entry>
      <entry key="ENS-Stub-For-FED-ADP">
       <bean class="mil.af.rl.phoenix.service.contexts.DispatchContainerContext">
        <property name="returnType" value="STUB" />
        <property name="nextEntityId" value="ENS-Stub-For-FED-ADP" />
       </bean>
      </entry>
     </map>
    </property>
   </bean>
  </constructor-arg>
 </bean>
</property>

 <property name="doServiceRegistration" ref="DO_SERVICE_REGISTRATION" />
 <property name="doBrokeringForServices" value="false" />
</bean>
```

3.5.1 Federation Adaptation Service Components

As any other Phoenix service, between the Federation Adaptation Service components we can find:

- the main service class: FederationAdaptationService
- context:
  - FederationAdaptationServiceContext

    - FederationAdaptationServiceContextInterface
- connectors:
    - RMIFederationAdaptationServiceConnector
    - RMIFederationAdaptationServiceConnectorInterface
- stubs:
    - RMIFederationAdaptationServiceStub
    - FederationAdaptationServiceStubInterface

In addition to these, the components which follow characterize just the Federation Adaptation Service:

- EventNotificationCallback
- Network monitoring using Mockets:
    - MocketStatusMonitor
    - MocketStatus
    - MocketStats
- State memory algorithm:
    - FederationAdaptationServiceWorker
    - BadState
    - BadStateSolution
    - BadStateSolutionsList
    - BadStatesList
    - ResumeSubsList

The FederationAdaptationService class manages the overall service. It implements the usual methods to start and stop the service and registers itself for monitor events in order to receive metric update notifications, handled by the EventNotificationCallback class.

When a new federate joins the federation, a new connection is established. This means that the service instantiates a byte channel with the new Federation Adaptation Service and it creates a MocketStatusMonitor object. This implements a thread which collects and analyzes the Mocket statistics and sends a notification via the byte channel to the corresponding federate in case of low-bandwidth situations.

It is also possible to specify that the Federation Adaptation Service does not have to perform network adaptation by disabling the PERFORM_ADAPT parameter in the spring configuration file. In such a case, the MocketStatusMonitor will not be initialized and the service will not collect mocket statistics coming from any other federates.

The classes that are grouped under “state memory algorithm” are responsible for the learning procedure that introduces hysteresis and prevents continuous vacillation from occurring between good and bad states.

Figure 18 shows the components involved in the network adaptation mechanisms performed by the Federation Adaptation Service. This particular example shows the simple case of two federates with a publish/subscribe relationship – with a single client that subscribes to a given type of information in Federate One, and a single client in Federate Two that publishes the

desired type of information. However, the same structure applies to multiple federates, given that each federation is treated as a pair-wise relationship. This approach should work unless federates share network links, and subscriptions between two federates take away from the bandwidth available between two other federates. Also, while this adaptation mechanism only addresses subscriptions, it would not be difficult to extend it in order to perform adaptations, for example in the case of query operations.

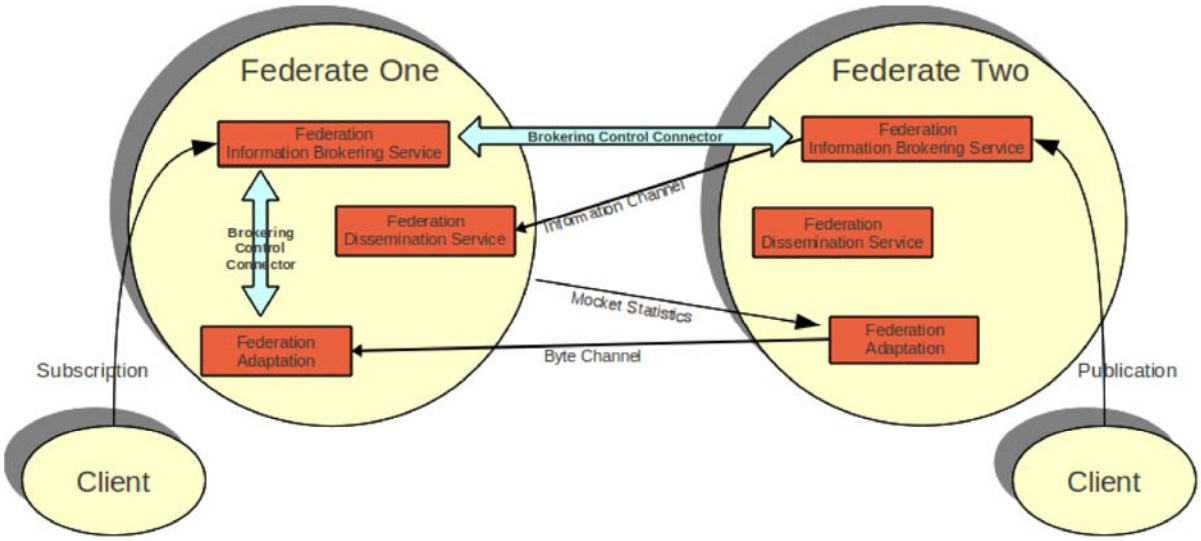

Figure 18: Components and Connections Related to Federation Adaptation Service

The overall operation of the Federation for publish / subscribe is described as follows. When the first client issues a subscription to the local Information Management System (IMS), the request is captured by the Federation Information Brokering Service (FIBS) via the Subscription Service. The FIBS retrieves the Remote Federation Service Proxy (RFSP) for Federate Two and uses it to remotely forward the received request for subscription. Once Federate Two obtains the request, the subscription is stored in a remote subscriptions table, ready to be matched against local publications.

When a client publishes information to the local IMS (Federate Two), such a publication is intercepted by the FIBS via Submission Service, which attempts to execute the predicate matching locally, by comparing the publication type and metadata with the remote subscriptions it may have previously stored in its remote subscription table. Publications for which the local matching succeeds are marked as matched, and sent to Federate One via the RFSP. Federate One receives the publication, verifies if it was already matched (and if it was not it matches it with the local subscriptions) and forwards it to the IMS. Finally the IMS takes care of the delivery to the correct subscriber clients.

The channel used to send publications from Federate Two to Federate One is marked as “Information channel” in Figure 18. If a network overload were to occur, this is the channel that would be transferring the majority of the data that would cause the overload. In order to monitor the behavior and performance of this channel, we use the Mockets communications library to perform the underlying transport. The channel is configured by the Federation Dissemination

Service (FDS) of the Federate One as shown in the following XML code, which reports the portion of the spring configuration file that defines the channel manager.

```
<property name="channelManager">
  <bean class="mil.af.rl.phoenix.service.channel.manager.BaseChannelManager">
    <constructor-arg>
      <bean class="mil.af.rl.phoenix.service.contexts.ChannelManagementContext">
        <property name="inputChannelMap">
          <map>
            ...
            <entry key="FederationDissemination.InputChannel.FromFedInfoBrokering">
              <bean class="mil.af.rl.phoenix.channel.ChannelContext">
                <property name="channelType" ref="CHANNEL_TYPE" />
                <property name="name" value="FederationDissemination.InputChannel.FromFedInfoBrokering" />
                <property name="applicationLevelContext">
                  <bean class="mil.af.rl.phoenix.channel.data.ApplicationLevelContext">
                    <property name="protocolId" value="serial" />
                  </bean>
                </property>
                <property name="endPointContext">
                  <bean class="mil.af.rl.phoenix.channel.EndPointContext">
                    <property name="hostAddress" ref="MY_IP" />
                    <property name="hostPort" ref="REMOTE_FED_DS_INPUT_PORT" />
                  </bean>
                </property>
                <property name="transportLevelContext">
                  <bean class="mil.af.rl.phoenix.channel.transport.TransportLevelContext">
                    <property name="protocolId" value="mocket" />
                  </bean>
                </property>
              </bean>
            </entry>
          </map>
        </property>
        ...
        <property name="inputManagerMap">
          <map>
            <entry key="FederationDissemination.InputMGR.FromFedInfoBrokering">
              <bean class="mil.af.rl.phoenix.service.managers.input.TimerBasedBufferInputManager">
                <constructor-arg>
                  <bean class="us.ihmc.infofed.federationdissemination.service.FederationDisseminationServiceWorker" />
                </constructor-arg>
                <constructor-arg value="MainFederationDisseminationService" />
                <constructor-arg value="FederationDissemination.InputMGR.FromFedInfoBrokering" />
              </bean>
            </entry>
            ...
          </map>
        </property>
        ...
        <property name="channelToManagerMap">
          <map>
            <entry key="FederationDissemination.InputMGR.FromFedInfoBrokering">
              <list>
                <value>FederationDissemination.InputChannel.FromFedInfoBrokering</value>
              </list>
            </entry>
            ...
          </map>
        </property>
      </bean>
    </constructor-arg>
  </bean>
</property>
```

The channel is connected to the FIBS of Federate Two during the federation establishment process, when the presence of Federate One is notified to the other federates (Federate Two in this example). Using the Mockets communications library in this case is necessary as it provides statistics about the messages sent and received, as well as information about the size of the outgoing message queue, the capacity of the link, the latency of the link, and the number of underlying transmission errors (as measured by the number of retransmissions). The following subsection provides details regarding monitoring the network channel using Mockets.

3.5.2 Network Monitoring Using Mockets

The Mockets communications library is used to provide the underlying transport service for the information channel that is used to transfer information between federates connected together. The main class performing network monitoring is Mocket Status Monitor (MSM). It is instantiated by the Federation Adaptation Service the first time there is a new connection due to a federate that joins the federation. The MSM is configured during initialization with the following parameters:

- The port used by the service to receive information from Mockets (Mocket Status Port);
- The minimum (MinPDS, MinRSDS) and maximum (MaxPDS, MaxRSDS) values representing the thresholds for the Mocket parameters Pending Data Size and Reliable Sequenced Data Size, necessary for the network monitoring;
- The IP address of the local federate;
- A reference to the byte channel just created to the new remote federate.

Every time a new federate joins the federation system the service simply adds the new byte channel to the Mocket Status Monitor instance to be able to send messages to it.

Mocket Status Monitor creates an object of the class Datagram Socket listening on the Mocket Status Port (default port is 1400) and periodically receives datagram packets containing two kinds of information about the Mocket channel:

- Endpoint information;
- Statistics information.

Endpoint information provides the local address and the local port which, usually are not very significant as are assigned by the system, and remote address and local port, which, instead are important as they allow the monitoring code to identify the channel to which the packet corresponds.

Statistics information are:

- The number of bytes transmitted
- The number of sent packets
- The number of retransmitted packets
- The number of bytes received
- The number of packets received
- The number of incoming packets that were discarded because they were duplicated
- The number of incoming packets that were discarded because there was no room to buffer them

- The number of incoming packets that are discarded because a message was not reassembled; this occurs when reassembly of a message from packet fragments is abandoned due to a timeout and the packets discarded are the fragments of the message that were received
- The size (in bytes) of the data that is queued in the pending packet queue awaiting transmission (Pending Data Size)
- The number of packets in the pending packet queue awaiting retransmission
- The size (in bytes) of the data that is in the reliable, sequenced packet queue awaiting acknowledgment (Reliable Sequenced Data Size)
- The number of packets in the reliable, sequenced packet queue awaiting acknowledgment
- The size (in bytes) of the data that is in the reliable, unsequenced packet queue awaiting acknowledgment
- The number of packets in the reliable, unsequenced packet queue awaiting acknowledgment.

The parameters that are important for the federation network monitoring are those indicated in the list as Pending Data Size and Reliable Sequenced Data Size.

When a new datagram packet with updated statistics information arrives, the Mocket Status Monitor checks the values of the two parameters and compares them with the thresholds. It could happen that, up until this moment, the network values were fine but now either one or both of the parameters are greater than the threshold: this means that the network is overloaded so a message is sent to the Federation Adaptation Service of the corresponding federate via a byte channel.

The other possibility is that the network was already overloaded but in the last update indicates that both Pending Data Size and Reliable Sequenced Data Size are lower than the thresholds. This means that there has been an improvement in the network channel, and a message is sent to the remote Federation Adaptation Service accordingly.

Note that these messages from the Mocket Status Monitor do not directly trigger adaptation of the subscriptions. Doing so might result in an unstable system that is continuously and unnecessarily adapting to changes. Instead, the input from the Mocket status Monitor is provided to the State Memory Algorithm, which decides on the actual adaption actions. The State Memory Algorithm is described in the following subsection.

3.5.3 State Memory Algorithm

The state Memory Algorithm introduces the hysteresis necessary into the adaptation mechanism of the federation service. The purpose of the hysteresis is to prevent unnecessary vacillations among different states due to borderline conditions and/or fluctuations in the environment.

The state memory algorithm has been realized in the FederationAdaptationServiceWorker class. The general structure of a worker in Phoenix is a class that manages channel inputs. It implements the interface ServiceWorkerInterface<T>, specifying the kind of channel has been used. In this case it is the byte channel that connects the two instances of the Federation Adaptation Service in the two federates (as shown in Figure 18). When Federate One receives

input from Federate Two on that channel, it means that the system, and in particular the connection between the two federates, is in a low-bandwidth situation.

The message that arrives contains the following information:

- Remote address of the federate that just sent the message
- A Boolean flag that indicates whether the federate needs to stop or to resume subscriptions
- The Pending Data Size
- The size (in bytes) of the data that is in the reliable, sequenced packet queue awaiting acknowledgment
- The Reliable Sequenced Data Size
- The number of packets in the reliable, sequenced packet queue awaiting acknowledgment
- The size (in bytes) of the data that is in the reliable, unsequenced packet queue awaiting acknowledgment
- The number of packets in the reliable, unsequenced packet queue awaiting acknowledgment.

Based on the value of the Boolean flag, it is possible to divide the algorithm in two parts: the first condition represents the situation where the network is overloaded and the system turned from a good state into a bad state. The second condition represents the case of an increase in bandwidth, which could imply that one or a few of the suspended subscriptions may be resumed.

The structure of the classes involved in the algorithm is shown in the UML schema of Figure 19. Bad State is the object that represents and collects all the information about a bad state of the system. It includes the ids of the subscriptions that are active at that moment, their priorities, the value of Pending Data Size and Reliable Sequenced Data Size, the number of times the system turned into that bad state and the list of solutions have been already tried.

Bad States List is a collection of all these bad states.

A solution, described by an object of the class Bad State Solution, is characterized by two lists of subscriptions: those that have been both stopped and those kept active, and by an instance of Resume Subs List, which collects the subscriptions that have been resumed.

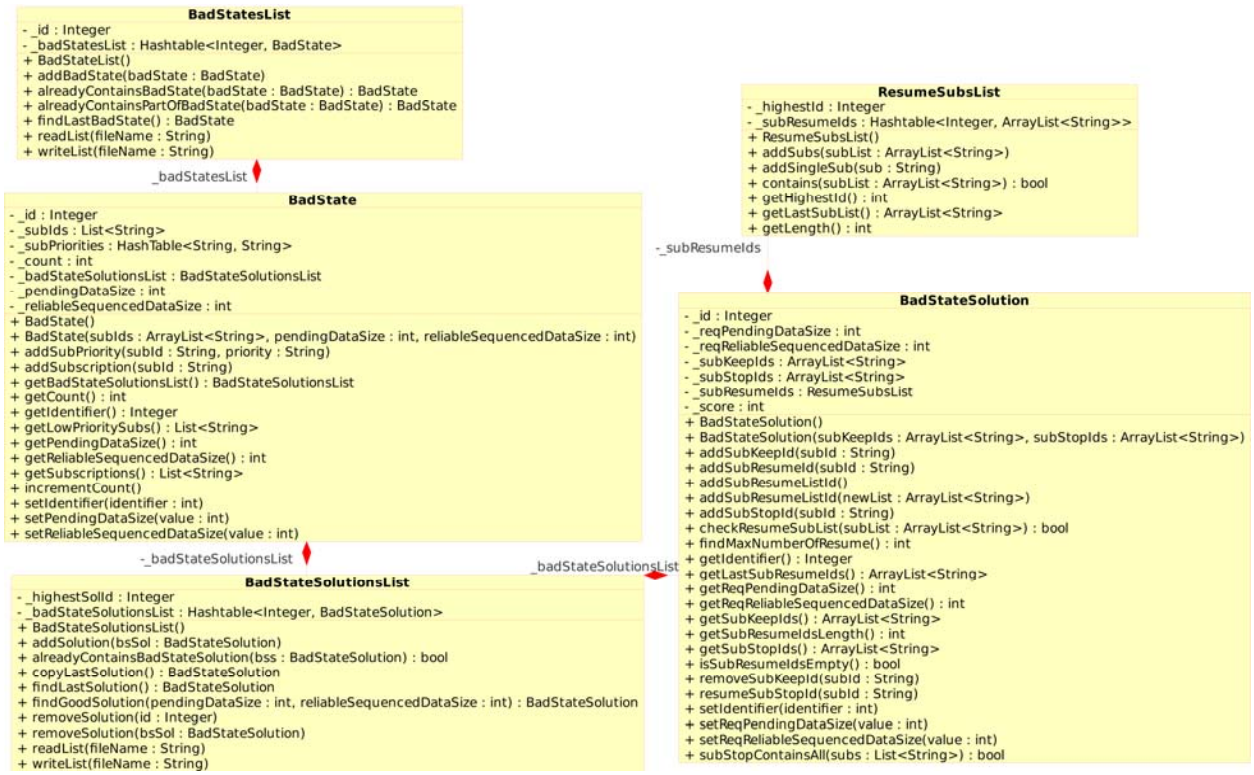

Figure 19: UML Diagram of Classes Used in the State Memory Algorithm

First Case: Network Overloaded

When the system registers a network overload, the system has entered a bad state. The Federation Adaptation Service examines the list of all the bad states previously recorded by the system, which could result in one of three possibilities:

1. The characteristics of the current bad state already occurred in the past, and was previously recorded;
2. A part of the current bad state is present;
3. The current bad state has not occurred in the past.

The first case means that the system has already been in that bad state sit uation so, at least a bad state solution for that already exists but none of the solutions were good enough. One possibility is to randomly search for another solution that has never been tried. After that the service can update the information of Pending Data Size and Reliable Sequenced Data Size if those stored in the bad state list are greater than those of the new bad state.

The second case, where the list already contains part of the new bad state, means that, even after disabling (some) low priority subscription(s), the current solution still represents a bad state. Therefore, the current solution needs to be further altered by disabling one or more additional low priority subscriptions that are still active, or changing the subscriptions that are disabled.

The third case means that this is the first time the system has entered into that bad state. To address this case, the adaptation service randomly chooses one or more low priority subscriptions to disable, which becomes the first solution for that bad state.

Once the subscriptions have been selected, the Federation Adaptation Service can invoke methods of the Federation Information Brokering Service, using the Brokering Control Connector, in order to suspend them.

Second Case: Bandwidth Increase

When the system registers an increase in bandwidth, it is possible that some of the suspended subscriptions could be resumed. The list of suspended subscriptions is stored into the last bad state with the solution that was determined for the bad state.

Two important parameters of each solution are the required pending data size (ReqPendingDataSize) and the required reliable sequenced data size (ReqReliableSequencedDataSize). These correspond to the maximum values allowed for these two network parameters while still maintaining good performance with this solution. Every time the system turns into that bad state and recognizes that as the good solution, the system needs to update those values if they are greater than the ones just measured for the network.

When the adaptation service receives a message indicating that the available bandwidth has increased, the first thing that it does is to check if a relationship exists between the subscriptions into the solution and those are effectively suspended. If the two sets are different those which exceed the solution are resumed and the algorithm ends.

Otherwise, one of many things could happen. The first case is determined by the number of elements in the resumed subscription list for the solution. If that list is empty it means that none of the suspended subscriptions have ever been restarted so we can choose one or more of them in a random way. On the other hand, if that list contains some of the subscription ids, we are probably creating a cycle and turning the system again into the same bad state. The list of the suspended subscriptions of the last solution may contain:

- just one element
- more than one element.

The first case provides no options because it means that the system is forced to create a cycle by resuming that subscription, as there are no other possibilities. The only thing the adaptation service can do is to restart that only if the ReqPendingDataSize and ReqReliableSequencedDataSize stored in the solution are greater than the values just measured in the system, otherwise it must simply wait.

The second case is better and it presents two different possibilities related to the number of possible combinations of resumed subscriptions. If this value is greater than the number of entries present in the resumed list in the solution, it means that the system still has a chance to restart suspended subscriptions without creating a cycle. One possibility is to randomly mix the stopped subscriptions until a new solution has been found.

On the other hand, if the system has already tried all the possible combination, this case is similar to the one described earlier, with just one subscription. Therefore, the system must wait until the values of ReqPendingDataSize and ReqReliableSequencedDataSize are greater than those just measured in the network prior to selecting a random subscription to resume.

Once the subscription(s) have been selected, if any, the Federation Adaptation Service can invoke methods of the Federation Information Brokering Service, using the Brokering Control Connector, in order to resume them.

3.6 Performance Evaluation

The Federation Service has been experimentally evaluated in terms of measuring overhead from adding federation capabilities to the base Information Management System (IMS). For this experimentation we considered both Apollo as well as Phoenix. We measured the performance of publish and subscribe operations considering a baseline installation of the evaluated IMS versus two installations of the same IMS sitting on two different nodes collaborating together through federation.

In order to understand the overhead that may be caused by the Federation Service, we measured the throughput in terms of time spent to send and receive information by the clients (execution time) and the maximum number of Information Objects per second that clients were able to send and receive (throughput).

Three different sets of experiments were conducted during the course of the project:

- Apollo and Apollo with Federation
- Apollo, Apollo with Federation, Phoenix (initial development verison), Phoenix with Federation
- Phoenix (Version 1.1.9), and Phoenix with Federation

While indirect comparisons are possible between Apollo and Phoenix by using these two independent sets of experiments, we did not have the opportunity to conduct a single experiment comparing all four configurations.

3.6.1 Experiments with Apollo and Federation

The first set of experiments compared Apollo with the initial version of Federation developed for Apollo. The experiment measured performance of publish and subscribe operations, with 1-4 publishers and 1-4 subscribers. Two different payload sizes were used, 0 KB (i.e., only metadata, no payload) and 10 KB (i.e., metadata + 10 KB payload). We measured both the time taken to publish as well as the latency of information delivery.

The experiments were performed on the MLAB testbed, consisting of 16 server nodes. Each node contained an Intel Celeron Processor at 2.66 GHz with 1 GB RAM, interconnected with 100 Mbps Fast Ethernet, running the Linux operating system

The test utilized was part of the standard Apollo benchmark suite, located in the package mil.af.rl.im.benchmark. This test publishes 1275 objects with varying payload sizes.

To measure the time taken to publish, we used the following configurations:

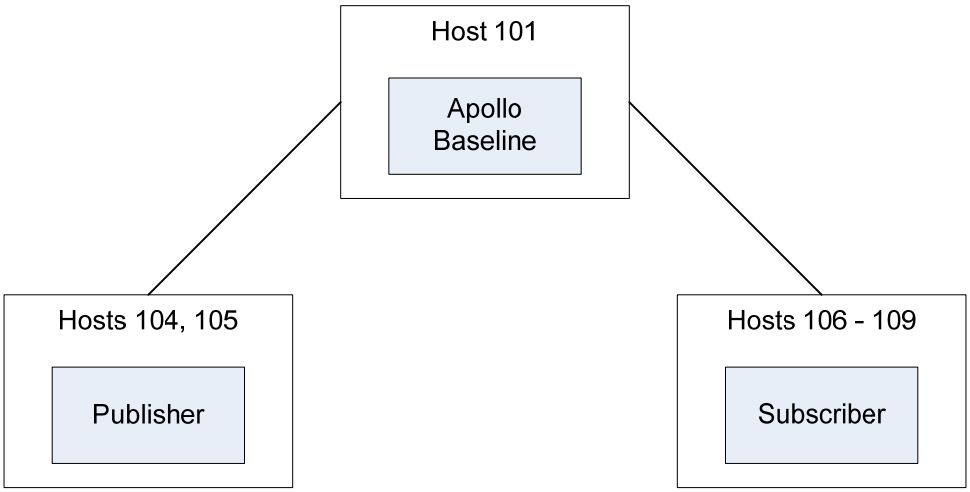

Figure 20: Configuration to Measure Baseline Performance of Apollo

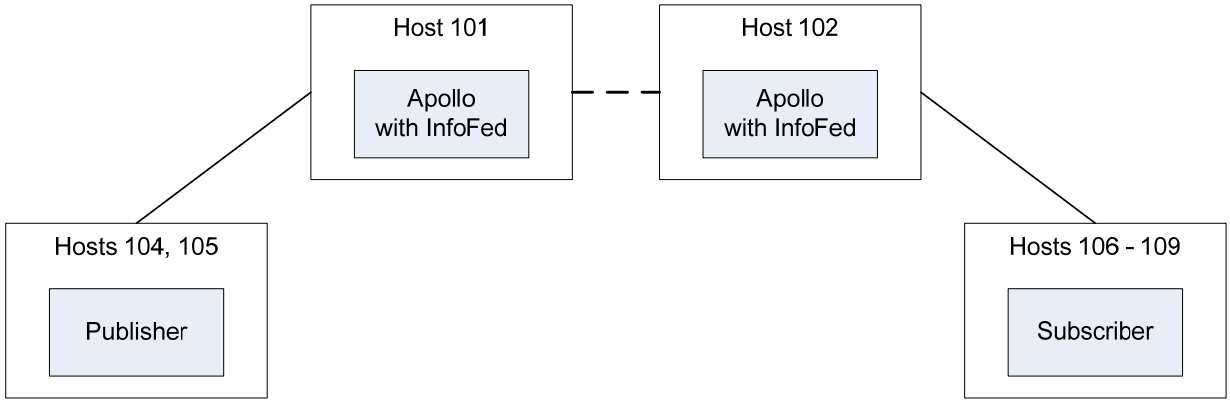

Figure 21: Configuration to Measure Performance of Two Federates with Apollo

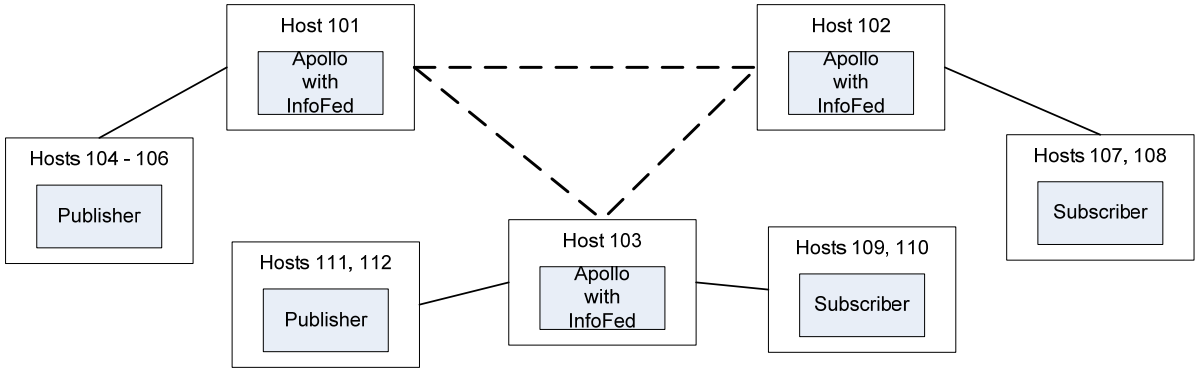

Figure 22: Configuration to Measure Performance of Three Federates with Apollo

To measure the latency of information delivery, we used the following configurations. Note that in this case, we host the publisher and subscriber on the same physical host to avoid any clock synchronization problems. That makes it simpler to measure the publish time and arrival time and measure the latency.

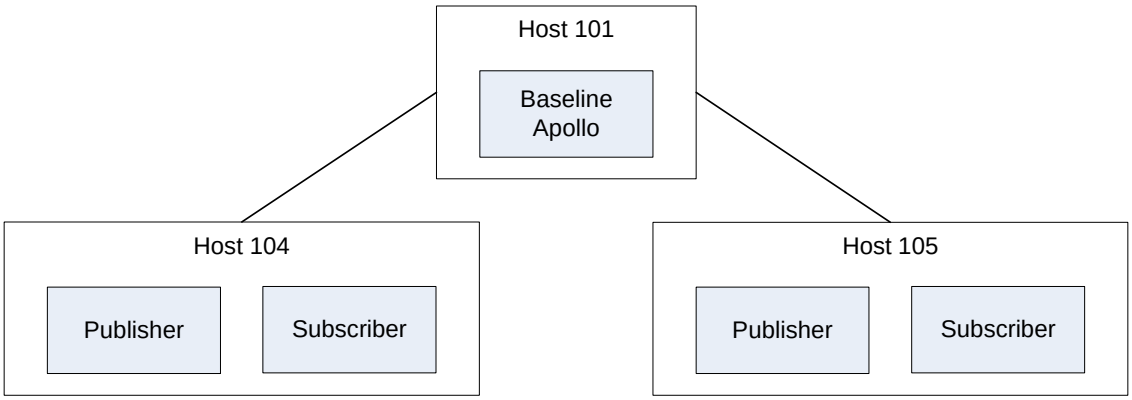

Figure 23: Configuration to Measure Latency with Baseline Apollo

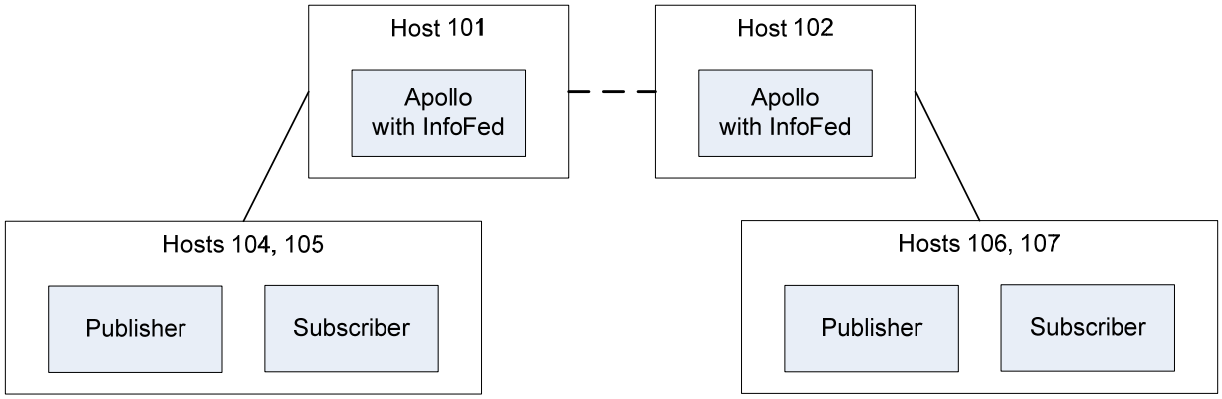

Figure 24: Configuration to Measure Latency with Two Federates and Apollo

The results from the experiments are shown in the tables below. As the results show, the performance of the publisher and subscriber actually improves with federation enabled. This counter-intuitive result can be explained by the distribution in processing load that occurs with federation enabled. Without federation, the entire processing load is placed on the single Apollo server instance. When federation is enabled, multiple instances of Apollo are used to serve different clients, with the resulting improvement in overall performance.

Table 1: Publisher Performance with Apollo and Federation

Publisher Performance (Time in Seconds to Publish 1275 Objects)						
# Clients	Apollo - 0KB	Apollo - 10KB	Two Federates - 0KB	Two Federates - 10KB	Three Federates - 0KB	Three Federates - 10bKB
2	32.04	38.87	29.85	31.12		
4	74.25	97.1	50.52	56.1	35.88	44.3
6	116.94	152.25	57.31	79.22	67.86	92.38
8	191.67	250.18	106.87	125.99	86.4	114.91

Table 2: Publisher Performance Improvement Factor with Apollo and Federation

Publisher Performance (Improvement!)				
# Clients	Two Federates - 0KB	Two Federates - 10KB	Three Federates - 0KB	Three Federates - 10bKB
2	1.07	1.25		
4	1.47	1.73	2.07	2.19
6	2.04	1.92	1.72	1.65
8	1.79	1.99	2.22	2.18

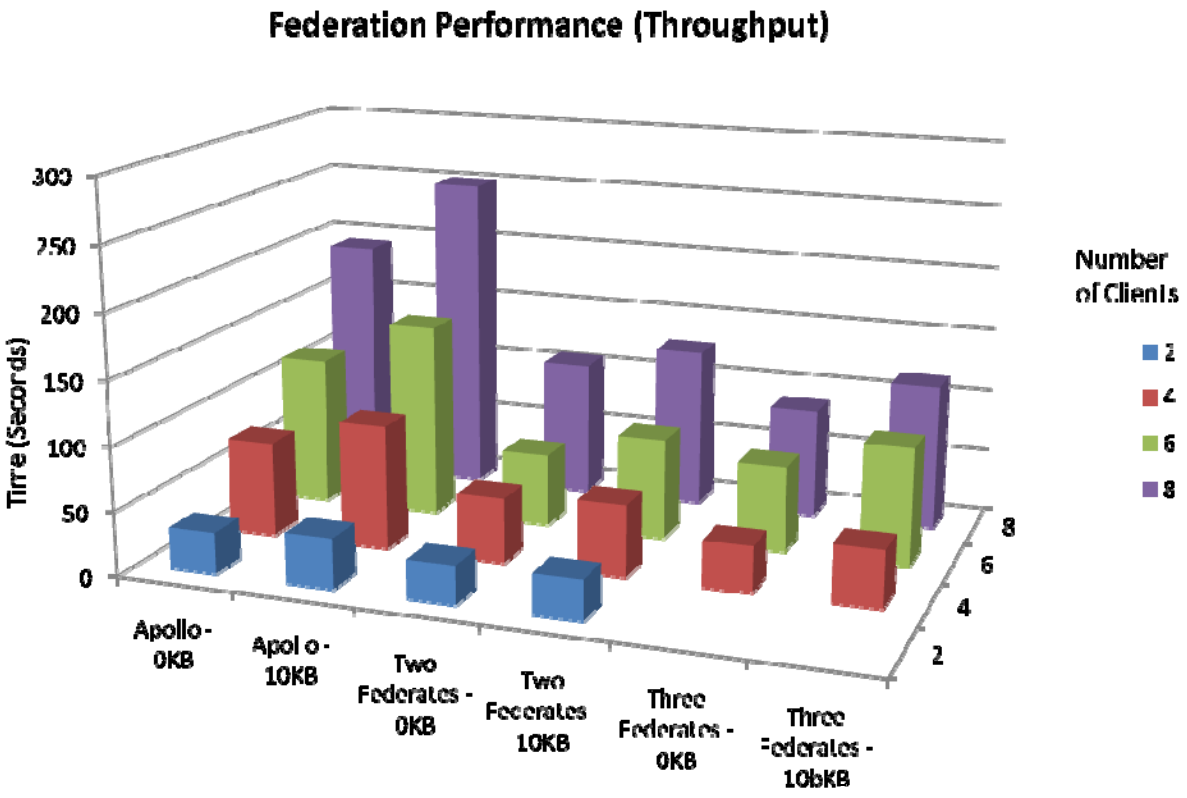

Figure 25: Publisher Performance with Apollo and Federation

Table 3: Subscriber Performance with Apollo and Federation

Subscriber Performance (Time in Seconds to Receive 1275 Objects)						
# Clients	Apollo - 0KB	Apollo - 10KB	Two Federates - 0KB	Two Federates - 10KB	Three Federates - 0KB	Three Federates - 10bKB
2	31.08	37.65	28.79	30.11		
4	38.76	49.43	25.78	28.8	23.16	26.83
6	42.71	51.98	24.38	24.9	27.13	34.13
8	50.42	64.18	30.28	33.56	23.29	30.84

Table 4: Subscriber Performance Improvement Factor with Apollo and Federation

Subscriber Performance (Improvement!)				
# Clients	Two Federates - 0KB	Two Federates - 10KB	Three Federates - 0KB	Three Federates - 10bKB
2	1.08	1.25		
4	1.50	1.72	1.67	1.84
6	1.75	2.09	1.57	1.52
8	1.67	1.91	2.16	2.08

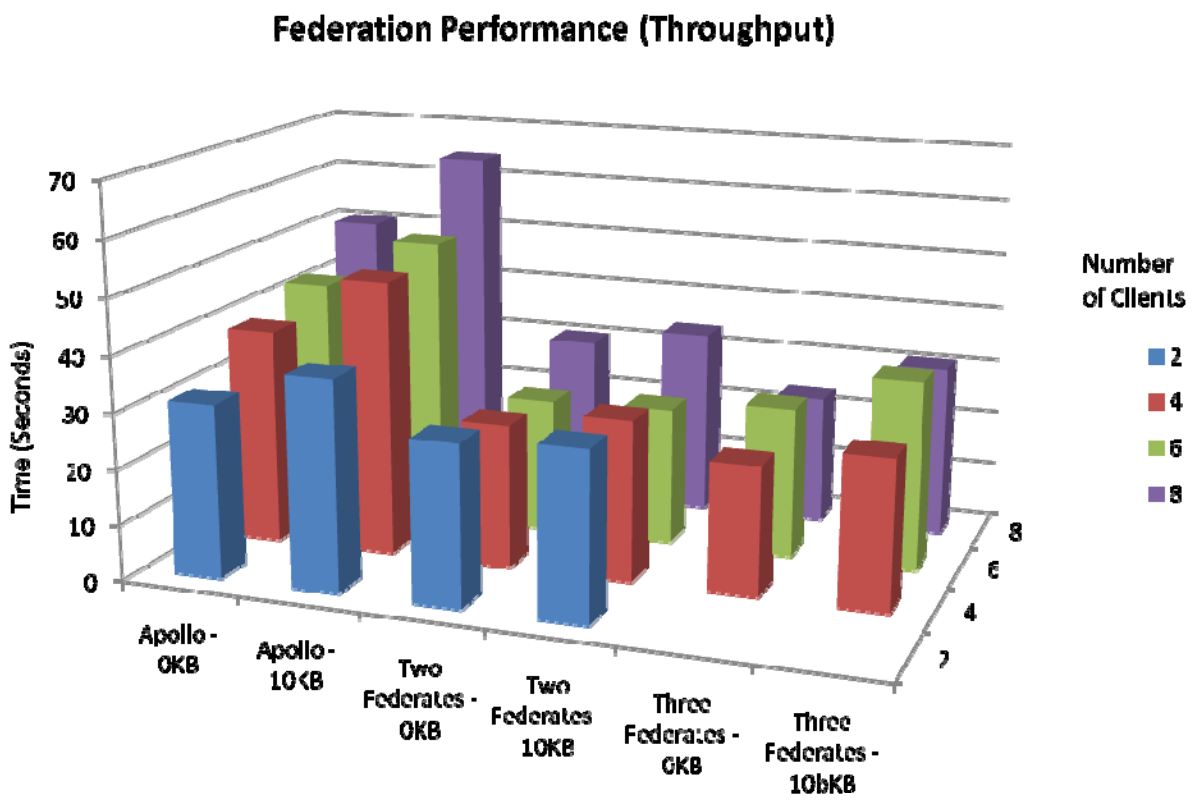

Figure 26: Subscriber Performance with Apollo and Federation

While the throughput increases with federation, there is an increase in latency of information delivery, caused by the extra instance(s) of Apollo injected into the overall system, as shown below. This latency is to be expected, given that any published information object has to traverse an additional network link, as well as another instance of the IM system. The increase in latency observed by our experimentation was relatively small (between 13.83%, and 36.28%), given that the experiments were conducted over high-speed Local Area Networks. In a deployment scenario where the federation occurs over long-haul links (e.g., Satellite), then the latency would be higher.

Table 5: Latency Measurements with Apollo and Federation

Subscriber - Latency			
Sleep Time (ms)	10	100	1000
Publisher - Baseline	41.61	154.58	1302.58
Publisher - Federated	39.44	153.69	1302.04
Subscriber - Baseline	24.21	21.79	22.35
Subscriber - Federated	27.56	26.40	30.46

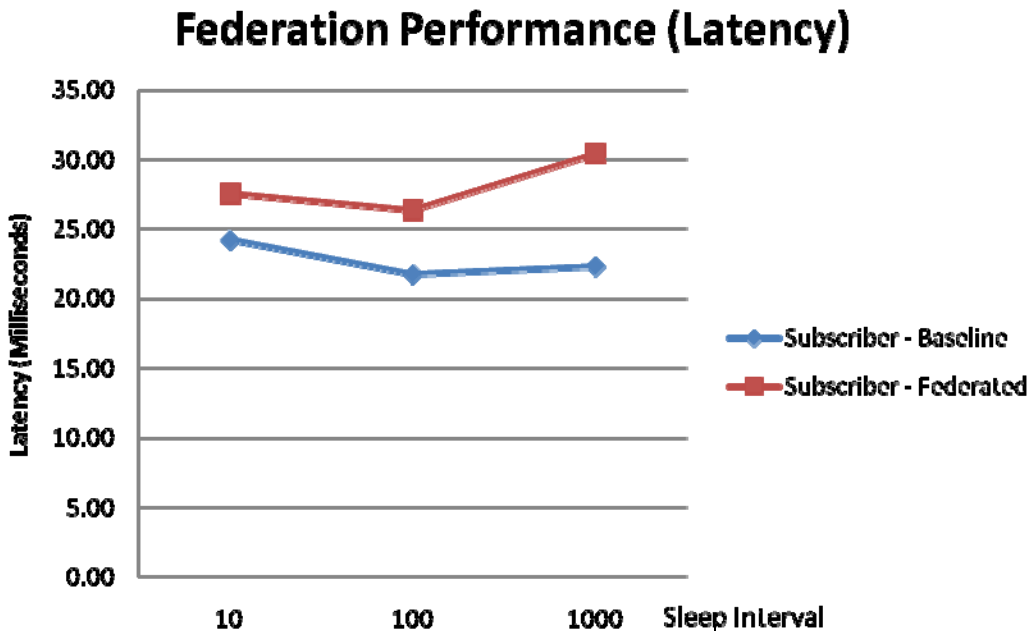

Figure 27: Latency of Information Delivery with Apollo and Federation

Overall, the results showed that latency increased slightly whereas throughout actually improved with the addition of Federation into Apollo.

3.6.2 Experiments with Apollo, Phoenix, and Federation

This experimental evaluation was conducted on virtual machines running on VMWare Server. The host machines have a dual core 3.06 GHz Intel Xeon processor and 4GB of memory. We deployed one virtual machine per physical machine. All the virtual machines were running Ubuntu Server 8.04, and were provided with 1GB of RAM.

All the tests involved one publisher client and two subscriber clients. In the first set, all are connected to the same instance of the IMS. In the second set of experiments, the publisher was connected to the first instance of the IMS and the subscribers were both connected to the second one, so the Information Objects were sent to the other side across the federation. The performance evaluation was executed using the benchmark suite provided with Apollo and adding clients that would support the information exchange protocols defined by the Phoenix architecture. We chose to run the clients with 55 iterations. With this configuration, publisher

and subscriber clients exchange 1275 Information Objects. Figure 28 shows the experimental scenario.

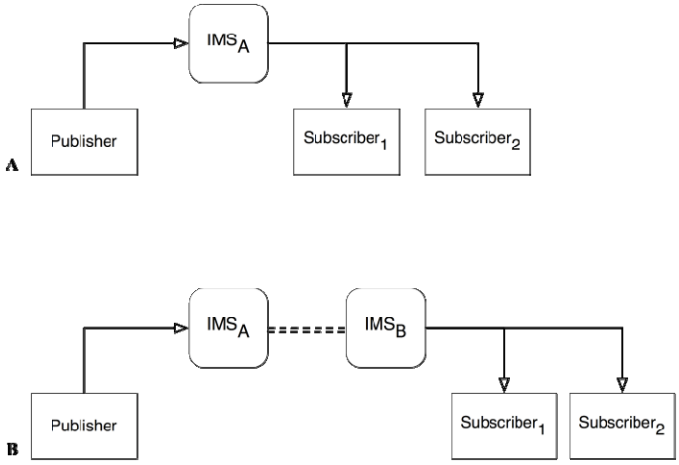

Figure 28: Experimental Scenario for the Performance Evaluation: The Baseline Version of the Tested IMS is shown in A. B shows the Configuration for the Tests with the Federation Service

The results are shown in the tables below:

Table 6: Time Spent to Publish and Receive Information Objects by Clients

Configuration	Publisher Time	Subscriber 1 Time	Subscriber 2 Time
Apollo Baseline	46.62 sec	41.16 sec	40.87 sec
Apollo with Federation	29.00 sec	28.14 sec	28.19 sec
Phoenix Baseline	3.81 sec	9.27 sec	9.26 sec
Phoenix with Federation	3.88 sec	4.81 sec	4.81 sec

Table 7: Throughput of Information Objects Published and Received by Clients

Configuration	Publisher	Subscriber 1	Subscriber 2
Apollo Baseline	29.2 IO/s	30.97 IO/s	31.19 IO/s
Apollo with Federation	42.10 IO/s	45.31 IO/s	45.22 IO/s
Phoenix Baseline	327.80 IO/s	137.55 IO/s	137.63 IO/s
Phoenix with Federation	322.09 IO/s	265.29 IO/s	265.29 IO/s

From the results presented in Table 6 and Table 7, we make two different observations. In the case of Apollo, we can see how the presence of the Federation Service improves the overall performance of the IMS instead of creating overhead. That actually makes sense: by adding federation capabilities, we split the load between the two federates (which are on separate physical nodes). In particular the publications are handled by the first instance of the IMS while the subscriptions are managed by the second instance.

This becomes even clearer when considering the numbers obtained in Phoenix tests. Phoenix uses asynchronous channels that rely on the Netty framework [8] for exchanging information from and to clients. On the publisher side, this means that the publication time and rate are not

affected by the computation that is necessary to manage every single piece of information being published. The publisher keeps putting information into the channel as fast as it can. The underlying layer will then manage the delivery to the IMS. This explains the very small difference in terms of performance of the publishing information to Phoenix with or without federation.

On the other hand, the subscribers' performance is affected by the computation the IMS needs to accomplish in order to manage the Information Objects it receives from the publisher and then dispatch them to the right subscriber clients. Time and reception rate are calculated from when the first piece of information is received to when the last one is delivered, which occurs concurrently with incoming information from publishers that needs to be handled. Having the load divided between two IMSs interconnected with federation shows its benefits also in the case of Phoenix.

The throughput results presented above show that from the client perspective, there is no performance degradation in terms of time and rate caused by adding federation capabilities to an IMS. The overhead of federation does manifest itself in terms of increased latency in information delivery. Latency of the information, i.e. the difference in time between when the information is produced by the publisher and when the same information is received by the subscriber, is crucial for certain types of applications, particularly in the tactical environment. The delay in the delivery of Information Objects to the subscribers clearly increases when such Information Objects have to be transmitted through the network to remote federates. Preliminary tests show that when the Federation Service is involved in the publish-delivery process, the latency of a single Information Object increases by 20% on average. This increase in latency is highly dependent on the network latency. As shown in Figure 28, there is an extra network hop involved with federation, which is the primary factor contributing to the latency.

One more noteworthy aspect with the results is the comparison between Apollo and Phoenix. The results show a slight improvement in the performance of federation between Apollo and Phoenix. When adapting our architecture for the Phoenix environment we started moving towards a lighter-weight services approach, and that seems to have produced benefits in terms of efficiency of the federation implementation. If we evaluate the subscriber side, Apollo with federation was about 1.4 times faster than Apollo baseline. Phoenix with federation instead is almost 2 times faster than Phoenix baseline.

3.6.3 Experiments with Phoenix and Federation

This final experimental evaluation was conducted using the latest version of Phoenix (1.1.9) made available to us towards the end of the project.

The hardware configuration utilized remains the same as the previous set of experiments. The evaluation was conducted on virtual machines running on VMWare Server. The host machines have a dual core 3.06 GHz Intel Xeon processor and 4GB of memory. We deployed one virtual machine per physical machine. All the virtual machines were running Ubuntu Server 8.04, and were provided with 1GB of RAM.

Like in the previous set of experiments, all the tests involved one publisher client and two subscriber clients. In the first set, all are connected to the same instance of the IMS. In the second set of experiments, the publisher was connected to the first instance of the IMS and the subscribers were both connected to the second one, so the Information Objects were sent to the other side across the federation. The performance evaluation was executed using the benchmark suite provided with Apollo and adding clients that would support the information exchange protocols defined by the Phoenix architecture. We chose to run the clients with 55 iterations. With this configuration, publisher and subscriber clients exchange 1275 Information Objects. Figure 28 showed the experimental scenario.

Time, Throughput, and Latency Results

The following tables and graphs show the performance measurements. The first set of three tables shows the performance of just the baseline Phoenix implementation. Each row of the table corresponds to running the benchmark test with a different number of iterations. For each iteration, the second column shows the number of objects that are actually published by the publisher client. For each subscriber client, we report on the time, the rate, and the latency of objects received. These are repeated with payloads of 0 bytes, 10 KB, and 100 KB.

Table 8: Performance of Baseline Phoenix with 0 KB Payload

		Publisher		First Subscriber						Second Subscriber					
				Subscription			Entire Communication			Subscription			Entire Communication		
ITER	N PUBL OBJECTS	TIME (sec)	RATE (obj/sec)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)
1	1	0.048	0.000	0.166	6.024	0.166	1.459	0.685	1.459	0.129	7.752	0.129	1.219	0.820	1.219
5	15	0.088	113.636	0.150	33.333	0.030	2.031	7.386	0.135	0.165	30.303	0.033	1.755	8.547	0.117
10	55	0.192	234.375	0.151	66.225	0.015	3.585	15.342	0.065	0.444	22.523	0.044	3.472	15.841	0.063
20	210	0.536	354.478	0.517	38.685	0.026	9.108	23.057	0.043	0.476	42.017	0.024	8.908	23.574	0.042
30	465	0.919	473.341	0.200	150.000	0.007	13.067	35.586	0.028	0.387	77.519	0.013	13.158	35.340	0.028
40	820	1.653	496.068	0.422	94.787	0.011	16.312	50.270	0.020	0.246	162.602	0.006	16.207	50.595	0.020
50	1275	2.299	532.840	0.262	190.840	0.005	19.690	64.754	0.015	0.313	159.744	0.006	19.666	64.833	0.015

Table 9: Performance of Baseline Phoenix with 10 KB Payload

		Publisher		First Subscriber						Second Subscriber					
				Subscription			Entire Communication			Subscription			Entire Communication		
ITER	N PUBL OBJECTS	TIME (sec)	RATE (obj/sec)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)
1	1	0.001	0.000	0.014	71.429	0.014	0.151	6.623	0.151	0.009	111.111	0.009	0.119	8.403	0.119
5	15	0.037	270.270	0.065	76.923	0.013	0.465	32.258	0.031	0.046	108.696	0.009	0.434	34.562	0.029
10	55	0.104	432.692	0.260	38.462	0.026	1.894	29.039	0.034	0.576	17.361	0.058	1.690	32.544	0.031
20	210	0.274	693.431	0.423	47.281	0.021	7.504	27.985	0.036	0.709	28.209	0.035	6.958	30.181	0.033
30	465	0.707	615.276	0.318	94.340	0.011	13.436	34.609	0.029	0.460	65.217	0.015	13.031	35.684	0.028
40	820	1.229	634.662	0.259	154.440	0.006	15.907	51.550	0.019	0.263	152.091	0.007	15.941	51.440	0.019
50	1275	2.112	580.019	0.202	247.525	0.004	18.475	69.012	0.014	0.238	210.084	0.005	18.442	69.136	0.014

Table 10: Performance of Baseline Phoenix with 100 KB Payload

		Publisher		First Subscriber						Second Subscriber					
				Subscription			Entire Communication			Subscription			Entire Communication		
ITER	N PUBL OBJECTS	TIME (sec)	RATE (obj/sec)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)
1	1	0.003	0.000	0.020	50.000	0.020	0.650	1.538	0.650	0.050	20.000	0.050	0.626	1.597	0.626
5	15	0.080	125.000	0.148	33.784	0.030	0.576	26.042	0.038	0.257	19.455	0.051	0.642	23.364	0.043
10	55	0.510	88.235	0.288	34.722	0.029	0.941	58.448	0.017	0.316	31.646	0.032	0.931	59.076	0.017
20	210	2.196	86.521	0.751	26.631	0.038	2.892	72.614	0.014	0.742	26.954	0.037	2.864	73.324	0.014
30	465	4.885	89.048	1.247	24.058	0.042	7.333	63.412	0.016	1.273	23.566	0.042	7.325	63.481	0.016
40	820	8.944	87.209	1.549	25.823	0.039	11.604	70.665	0.014	1.580	25.316	0.040	11.623	70.550	0.014
50	1275	14.073	87.046	1.930	25.907	0.039	17.568	72.575	0.014	1.973	25.342	0.039	17.598	72.451	0.014

The following set of three tables show the performance of Phoenix with federation. As before, we report on the results for the publisher, the first subscriber, and the second subscriber. The payloads are also the same as before – 0 KB for the first table, 10 KB for the second table, and 100 KB for the third table.

Table 11: Performance of Phoenix with Federation with 0 KB Payload

		Publisher		First Subscriber						Second Subscriber					
				Subscription			Entire Communication			Subscription			Entire Communication		
ITER	N PUBL OBJECTS	TIME (sec)	RATE (obj/sec)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)
1	1	0.042	0.000	0.139	7.194	0.139	2.096	0.477	2.096	0.158	6.329	0.158	2.077	0.481	2.077
5	15	0.106	94.340	0.076	65.789	0.015	2.671	5.616	0.178	0.218	22.936	0.044	2.660	5.639	0.177
10	55	0.278	161.871	0.679	14.728	0.068	4.363	12.606	0.079	0.271	36.900	0.027	4.315	12.746	0.078
20	210	0.773	245.796	0.363	55.096	0.018	9.666	21.726	0.046	0.728	27.473	0.036	9.565	21.955	0.046
30	465	2.267	191.884	0.216	138.889	0.007	13.569	34.269	0.029	0.198	151.515	0.007	13.445	34.585	0.029
40	820	5.794	134.622	0.302	132.450	0.008	16.728	49.020	0.020	0.498	80.321	0.012	16.532	49.601	0.020
50	1275	9.332	131.269	0.175	285.714	0.003	19.841	64.261	0.016	0.307	162.866	0.006	19.693	64.744	0.150

Table 12: Performance of Phoenix with Federation with 10 KB Payload

		Publisher		First Subscriber						Second Subscriber					
				Subscription			Entire Communication			Subscription			Entire Communication		
ITER	N PUBL OBJECTS	TIME (sec)	RATE (obj/sec)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)
1	1	0.044	0.000	0.033	30.303	0.033	2.190	0.457	2.190	0.116	8.621	0.116	2.255	0.443	2.255
5	15	0.119	84.034	0.106	47.170	0.021	3.422	4.383	0.228	0.166	30.120	0.033	3.254	4.610	0.217
10	55	0.307	146.580	0.242	41.322	0.024	4.998	11.004	0.091	0.328	30.488	0.033	4.868	11.298	0.089
20	210	0.966	196.687	0.412	48.544	0.021	10.689	19.646	0.051	0.799	25.031	0.040	10.968	19.147	0.052
30	465	3.246	134.011	0.489	61.350	0.016	17.601	26.419	0.038	0.913	32.859	0.030	16.840	27.613	0.036
40	820	7.089	110.030	0.245	163.265	0.006	21.226	38.632	0.026	0.257	155.642	0.006	20.700	39.614	0.025
50	1275	10.875	112.644	0.149	335.570	0.003	23.825	53.515	0.019	0.308	162.338	0.006	23.601	54.023	0.019

Table 13: Performance of Phoenix with Federation with 100 KB Payload

		Publisher		First Subscriber						Second Subscriber					
				Subscription			Entire Communication			Subscription			Entire Communication		
ITER	N PUBL OBJECTS	TIME (sec)	RATE (obj/sec)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)	TIME (sec)	RATE (obj/sec)	LATENCY (sec/obj)
1	1	0.052	0.000	0.053	18.868	0.053	0.944	1.059	0.944	0.034	29.412	0.034	0.941	1.063	0.941
5	15	0.131	76.336	0.585	8.547	0.117	1.016	14.764	0.068	0.563	8.881	0.113	1.006	14.911	0.067
10	55	0.513	87.719	0.627	15.949	0.063	1.319	41.698	0.024	0.672	14.881	0.067	1.350	40.741	0.025
20	210	1.937	98.090	1.975	10.127	0.099	4.290	48.951	0.020	1.973	10.137	0.099	4.282	49.043	0.020
30	465	4.346	100.092	6.210	4.831	0.207	22.691	20.493	0.049	6.207	4.833	0.207	22.684	20.499	0.049
40	820	7.417	105.164	6.937	5.766	0.173	61.500	13.333	0.075	6.931	5.771	0.173	61.503	13.333	0.075
50	1275	10.810	113.321	8.950	5.587	0.179	113.768	11.207	0.089	8.945	5.590	0.179	113.757	11.208	0.089

The following graphs compare the performance of baseline Phoenix and Phoenix with Federation. We compare publication time, publication rate, subscription time, subscription rate. For the subscription, we provide results for each of the two subscribers. Independent graphs are shown for each payload (0 KB, 10 KB, and 100 KB).

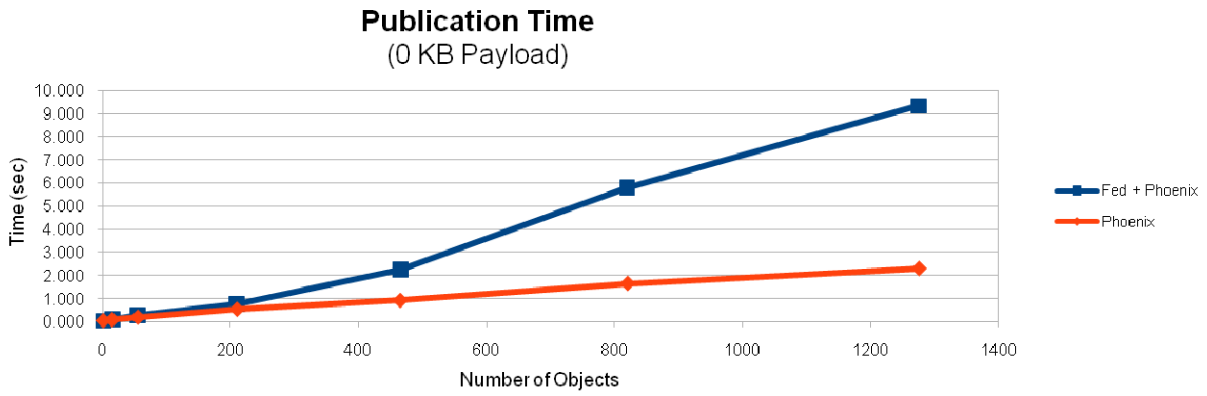

Figure 29: Comparison of Publication Time with 0 KB Payload

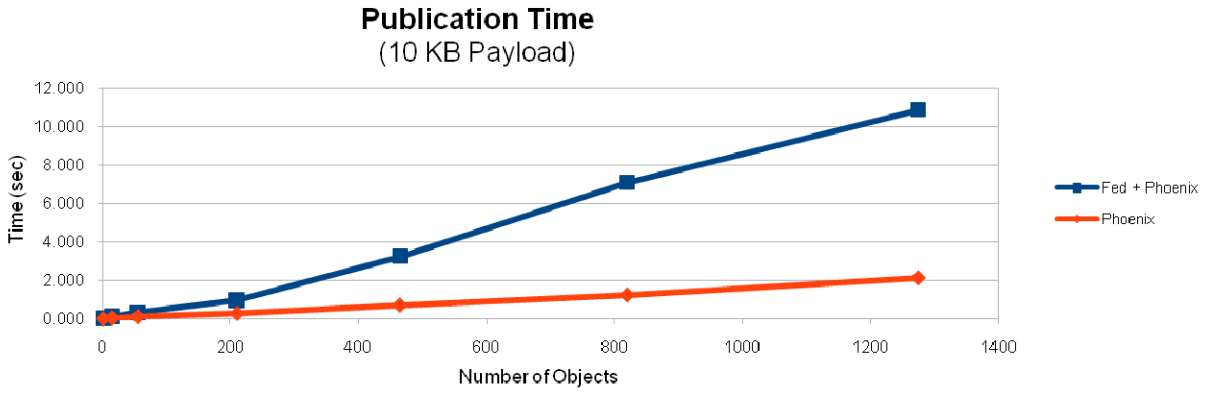

Figure 30: Comparison of Publication Time with 10 KB Payload

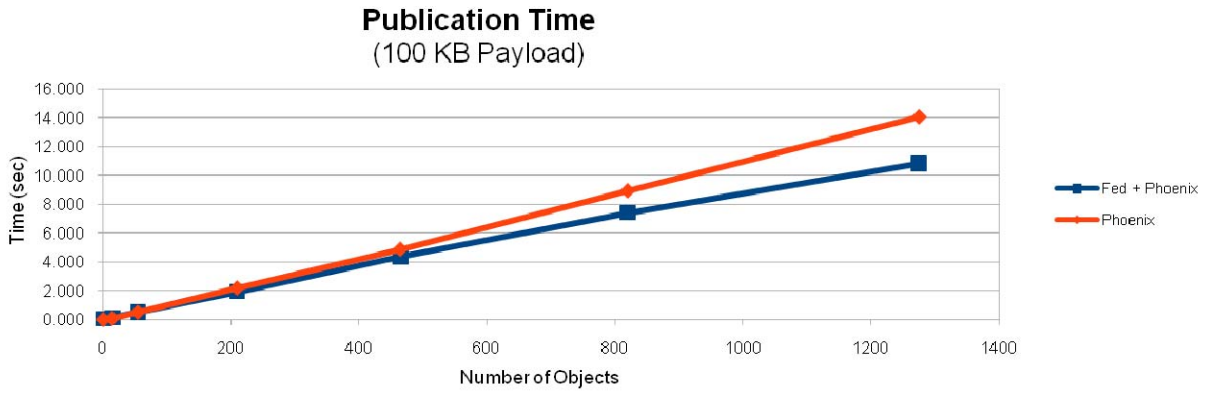

Figure 31: Comparison of Publication Time with 100 KB Payload

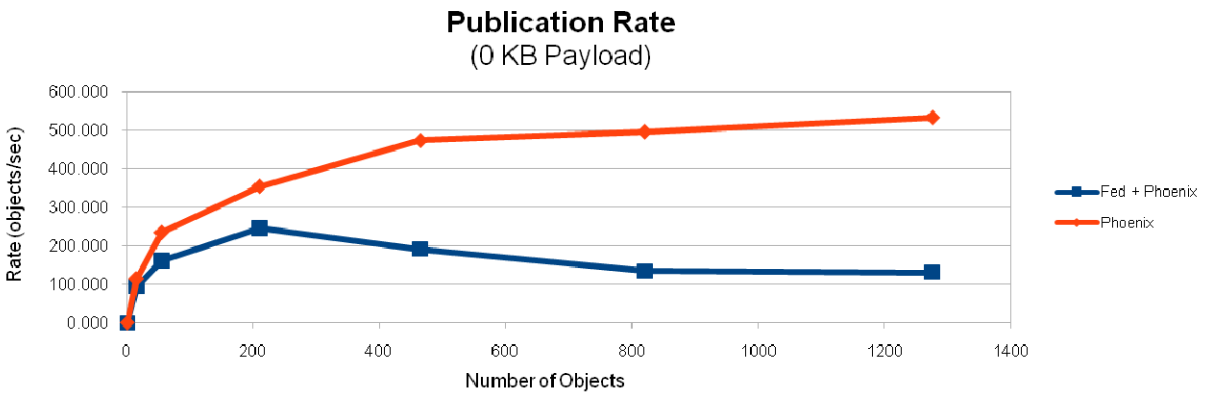

Figure 32: Comparison of Publication Rate with 0 KB Payload

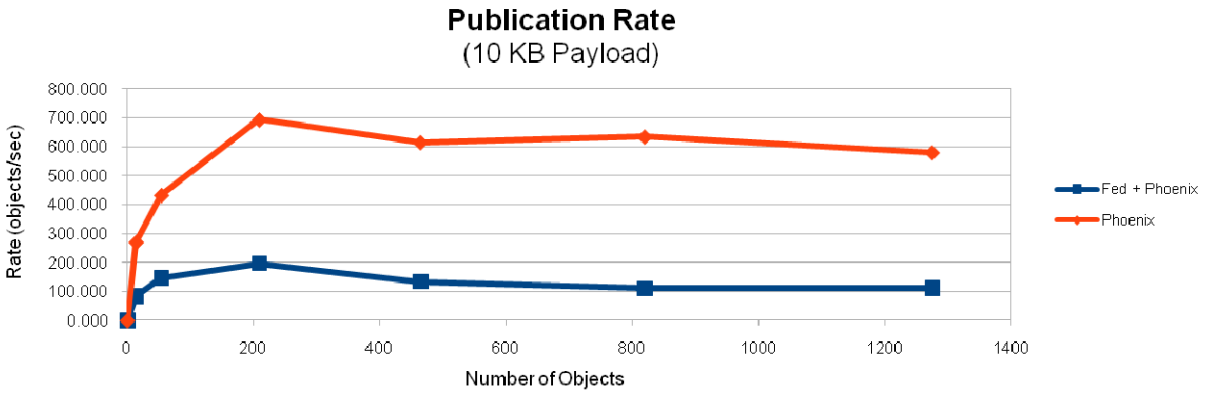

Figure 33: Comparison of Publication Rate with 10 KB Payload

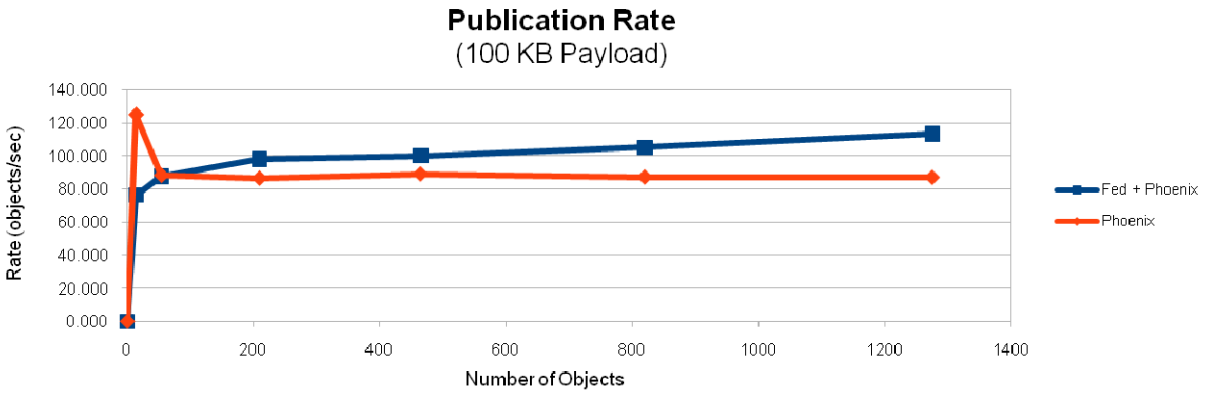

Figure 34: Comparison of Publication Rate with 100 KB Payload

Figure 35: Comparison of Subscriber Time with 0 KB Payload

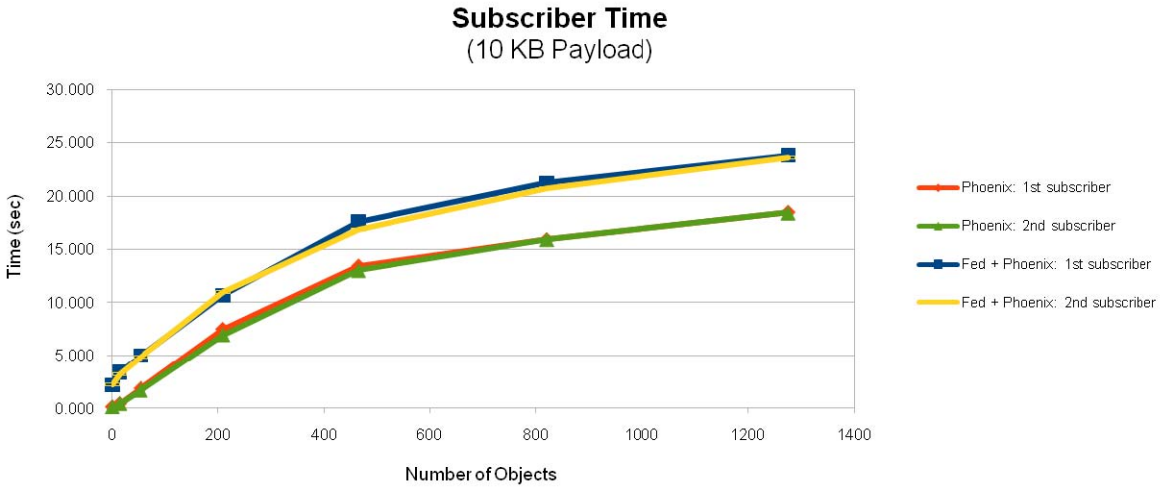

Figure 36: Comparison of Subscriber Time with 10 KB Payload

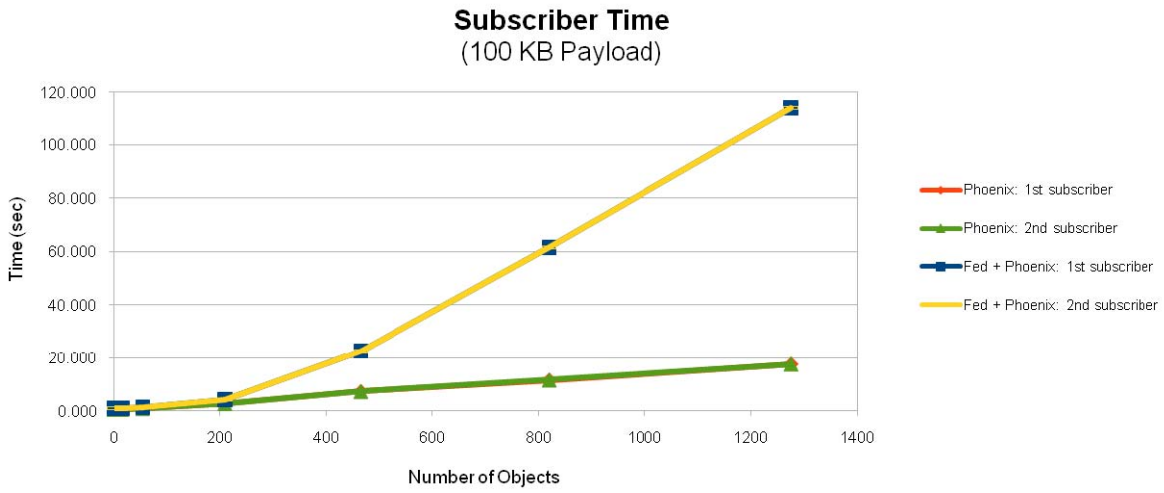

Figure 37: Comparison of Subscriber Time with 100 KB Payload

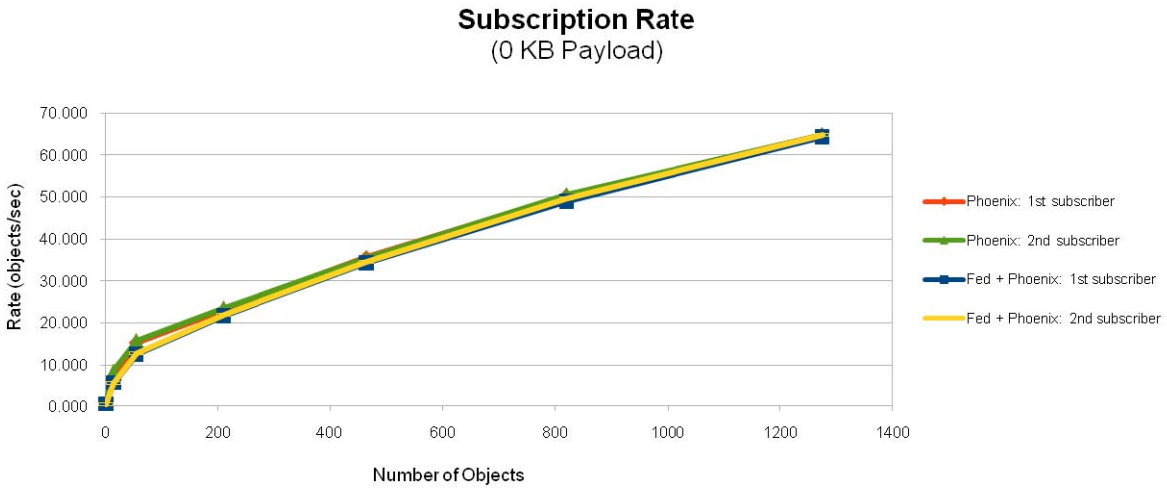

Figure 38: Comparison of Subscription Rate with 0 KB Payload

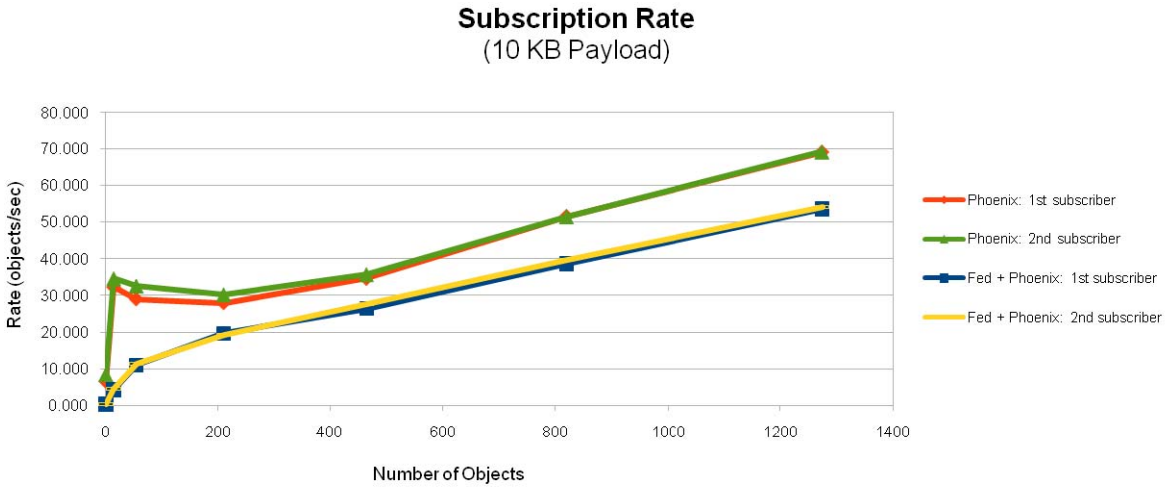

Figure 39: Comparison of Subscription Rate with 10 KB Payload

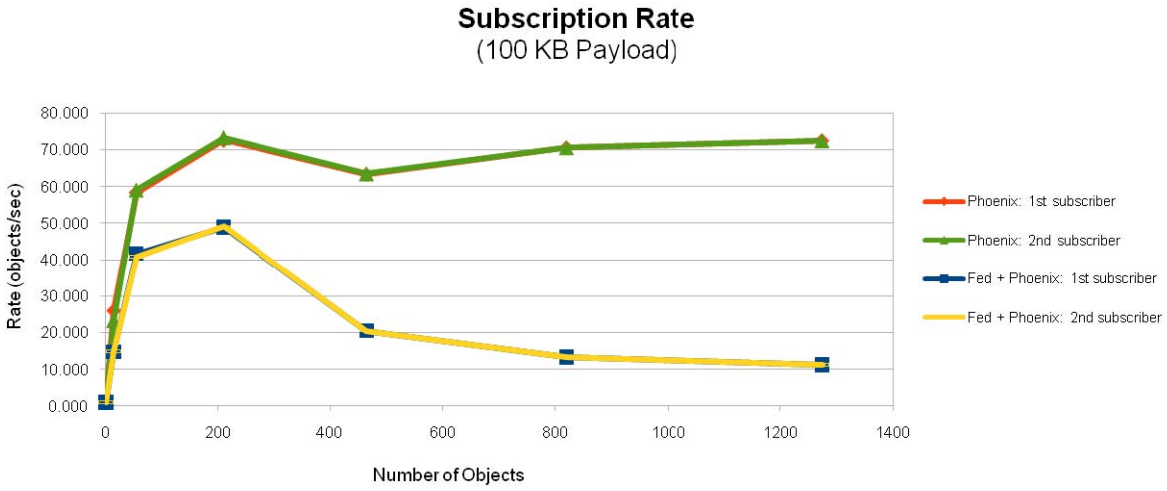

Figure 40: Comparison of Subscription Rate with 100 KB Payload

The results show that the publication times and rates are slower as a result of the additional overhead introduced by federation. This result differs from the Apollo case, where publication was actually faster with federation. There was insufficient time in the project to further analyze this result. The results also show that the subscription times and rates are slower with federation. This decrease in performance is caused by the need to transmit the published object and payload an extra hop over the network to the destination node.

The next set of results compare the average latency of arrival of objects at the subscribers. Since the latency of the first object is always high, including that data point tends to make the other comparisons more difficult (as seen in Figure 41). Therefore, we show the results without the single object case in the following three graphs, one for each payload configuration.

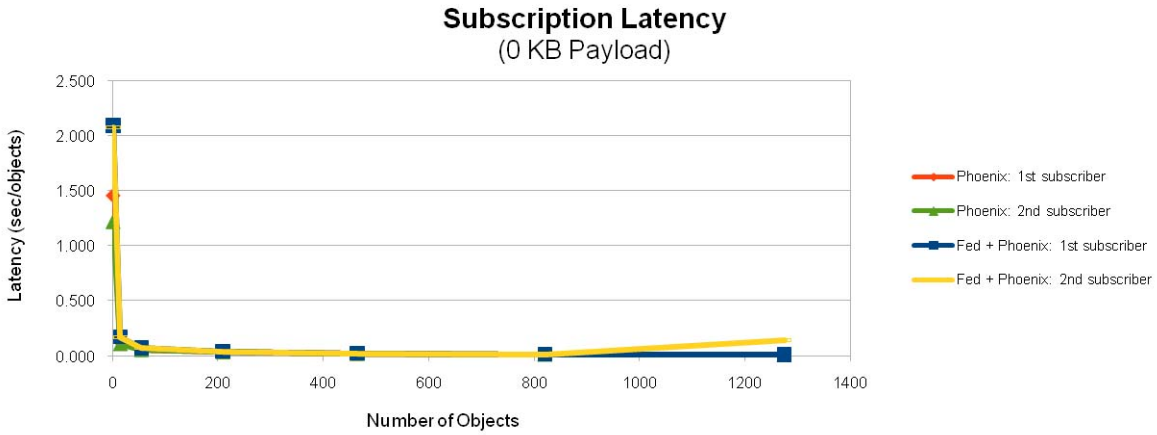

Figure 41: Average Latency of Objects Received by Subscribers with 0 KB Payload (Including Single Object Case)

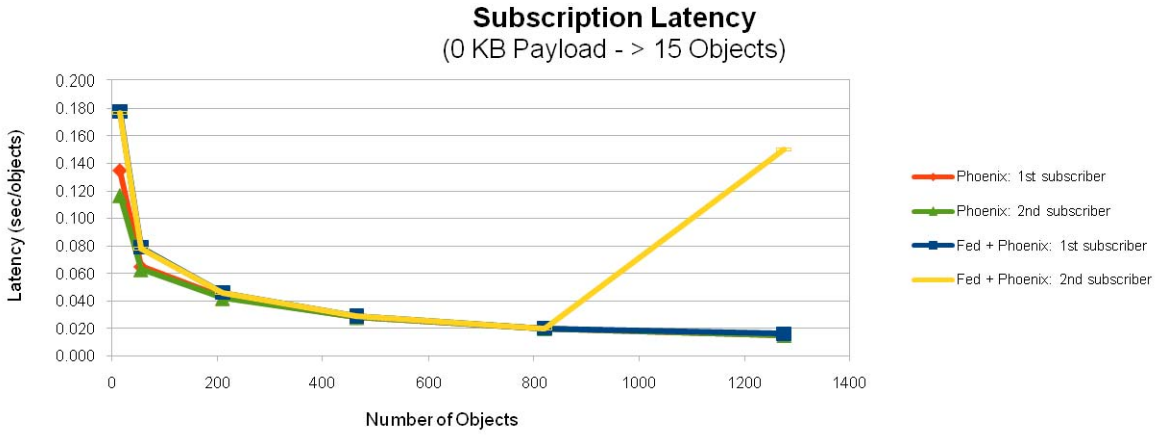

Figure 42: Average Latency of Objects Received by Subscribers with 0 KB Payload

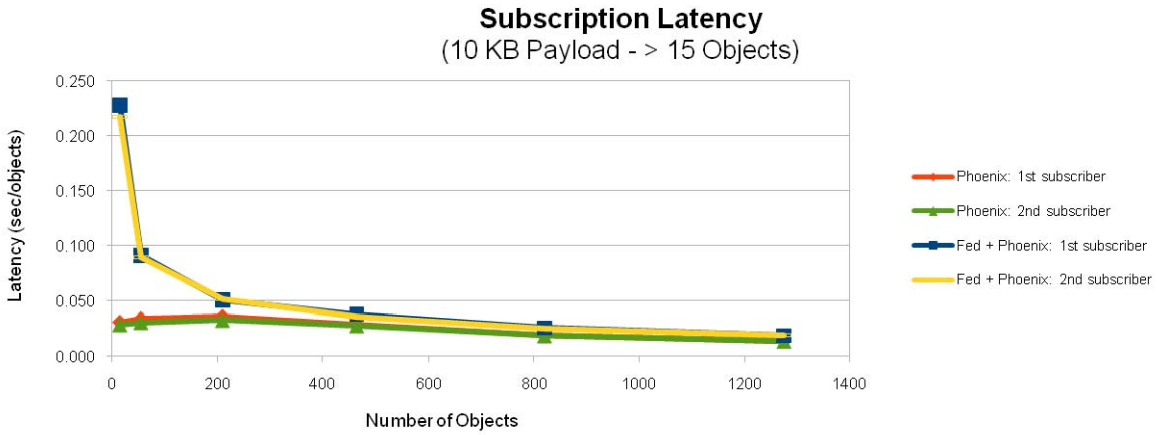

Figure 43: Average Latency of Objects Received by Subscribers with 10 KB Payload

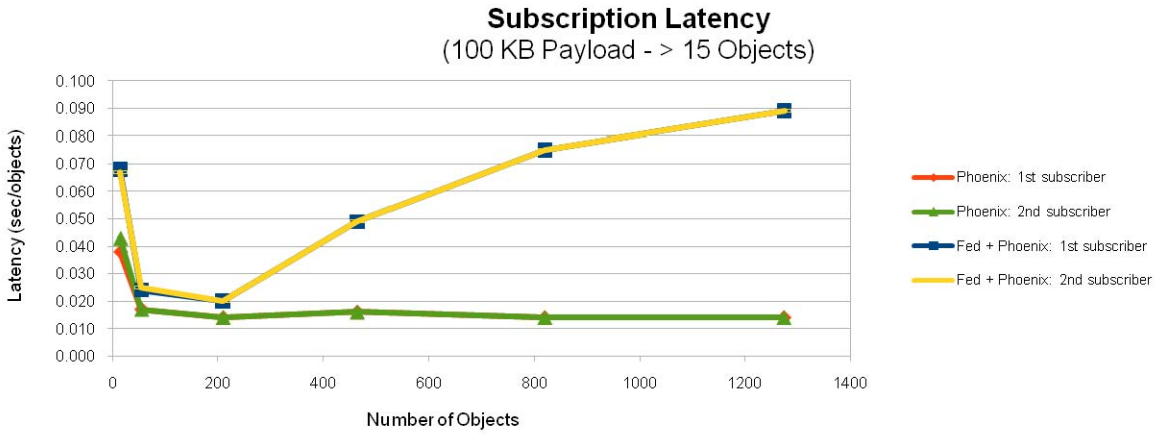

Figure 44: Average Latency of Objects Received by Subscribers with 100 KB Payload

The results show that from a latency perspective, the latency increases slightly with federation as a function of the payload size. With a payload size of 0 KB or 10 KB, there is virtually no increase in the observed latency of information delivery. With a payload size of 100 KB, there is an increase, which is caused by the extra network link that must be traversed in the case of federation. Note that these results are over a 100 Mbps wired network. With a constrained network, the latency is likely to be higher even with smaller payloads.

Bandwidth Results

The next set of results report on the bandwidth consumed in performing the above publish and subscribe experiments. The following tables show the results for Phoenix and Phoenix and Federation, broken down into the different communication pairs (e.g., Publisher to IMS, IMS to Subscriber, IMS to IMS, etc.) As before, we provide results for payloads of 0 KB, 10 KB, and 100 KB.

Table 14: Bandwidth Utilization for Phoenix with 0 KB Payload

FROM	TO	PACKETS	TIME (sec)	AVG. PACKET/SEC	BYTES	AVG. BYTE/SEC
publisher	phoenix	974	100.632	9.679	1260142	12522.279
phoenix	publisher	761	100.665	7.560	50234	499.022
phoenix	1st subscriber	1174	104.233	11.263	1142017	10956.386
1st subscriber	phoenix	987	104.223	9.470	65150	625.102
phoenix	2nd subscriber	1164	104.251	11.165	1141357	10948.164
2nd subscriber	phoenix	960	104.251	9.209	63368	607.841
	TOTAL	6020			3722268	

Table 15: Bandwidth Utilization for Phoenix with Federation with 0 KB Payload

FROM	TO	PACKETS	TIME (sec)	AVG. PACKET/SEC	BYTES	AVG. BYTE/SEC
publisher	1st federate	990	113.925	8.690	1261850	11076.147
1st federate	publisher	770	113.924	6.759	54240	476.107
1st federate	2nd federate	1653	202.256	8.173	1702546	8417.777
2nd federate	1st federate	692	202.256	3.421	67106	331.787
2nd federate	1st subscriber	1252	103.619	12.083	1485040	14331.735
1st subscriber	2nd federate	985	103.619	9.506	65018	627.472
2nd federate	2nd subscriber	1247	103.649	12.031	1484710	14324.403
2nd subscriber	2nd federate	977	103.649	9.426	64490	622.196
	TOTAL	8566			6185000	

Table 16: Bandwidth Utilization for Phoenix with 10 KB Payload

FROM	TO	PACKETS	TIME (sec)	AVG. PACKET/SEC	BYTES	AVG. BYTE/SEC
publisher	phoenix	9897	113.559	87.153	14906335	131265.113
phoenix	publisher	1793	113.558	15.789	118346	1042.163
phoenix	1st subscriber	10106	105.179	96.084	14788804	140606.053
1st subscriber	phoenix	3924	105.180	37.307	258992	2462.369
phoenix	2nd subscriber	10096	105.090	96.070	14788144	140718.851
2nd subscriber	phoenix	3646	105.090	34.694	240644	2289.885
	TOTAL	39462			45101265	

Table 17: Bandwidth Utilization for Phoenix with Federation with 10 KB Payload

FROM	TO	PACKETS	TIME (sec)	AVG. PACKET/SEC	BYTES	AVG. BYTE/SEC
publisher	1st federate	9914	102.646	96.584	14908109	145238.090
1st federate	publisher	1705	102.780	16.589	115950	1128.138
1st federate	2nd federate	11666	109.397	106.639	15376199	140554.119
2nd federate	1st federate	650	109.336	5.945	60644	554.657
2nd federate	1st subscriber	10184	104.737	97.234	15131827	144474.512
1st subscriber	2nd federate	3879	104.737	37.036	256022	2444.427
2nd federate	2nd subscriber	10177	104.740	97.164	15131365	144465.963
2nd subscriber	2nd federate	4014	104.740	38.323	264932	2529.425
	TOTAL	52189			61245048	

Table 18: Bandwidth Utilization for Phoenix with 100 KB Payload

FROM	TO	PACKETS	TIME (sec)	AVG. PACKET/SEC	BYTES	AVG. BYTE/SEC
publisher	phoenix	90134	125.299	719.351	136397428	1088575.551
phoenix	publisher	18454	125.298	147.281	1218432	9724.273
phoenix	1st subscriber	90623	118.244	766.407	136651020	1155669.801
1st subscriber	phoenix	22358	118.244	189.084	1477632	12496.465
phoenix	2nd subscriber	90422	118.112	765.562	136207376	1153205.229
2nd subscriber	phoenix	25082	118.112	212.358	1657856	14036.305
	TOTAL	337073			413609744	

Table 19: Bandwidth Utilization for Phoenix with Federation with 100 KB Payload

FROM	TO	PACKETS	TIME (sec)	AVG. PACKET/SEC	BYTES	AVG. BYTE/SEC
publisher	1st federate	91062	121.932	746.826	137761912	1129825.739
1st federate	publisher	197748	121.931	1621.802	1314084	10777.276
1st federate	2nd federate	99513	237.523	418.962	141433292	595450.933
2nd federate	1st federate	4801	237.461	20.218	439363	1850.253
2nd federate	1st subscriber	91787	233.087	393.789	138021452	592145.645
1st subscriber	2nd federate	42332	233.371	181.394	2794444	11974.256
2nd federate	2nd subscriber	91768	233.035	393.795	138004270	592204.047
2nd subscriber	2nd federate	33418	233.035	143.403	2206508	9468.569
	TOTAL	652429			561975325	

The following graphs compare the bandwidth utilization of Phoenix and Phoenix with Federation. As is to be expected, the bandwidth utilized is higher given the extra link that must be traversed by the act of federating two instances of Phoenix.

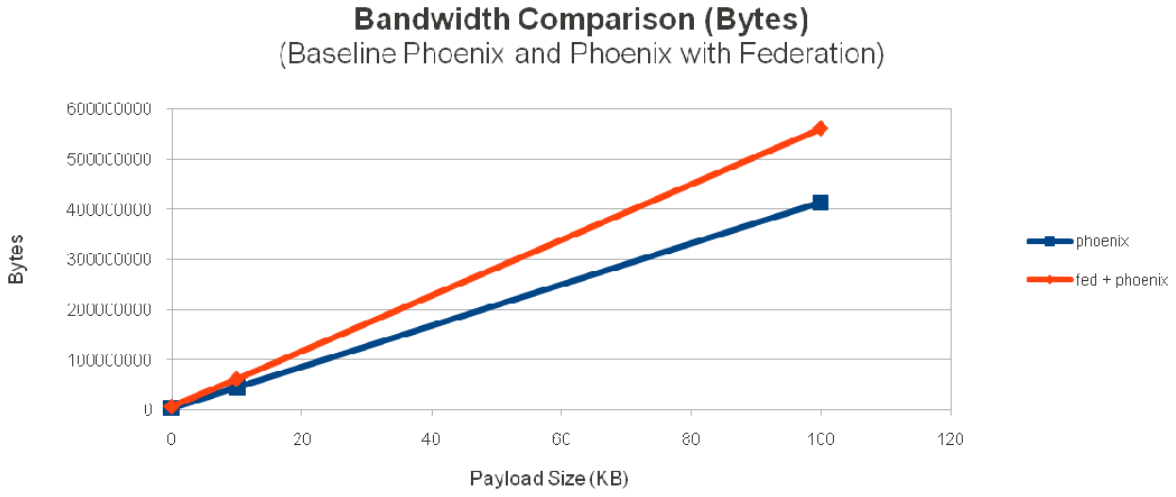

Figure 45: Bandwidth Comparison (Bytes) of Baseline Phoenix and Federation

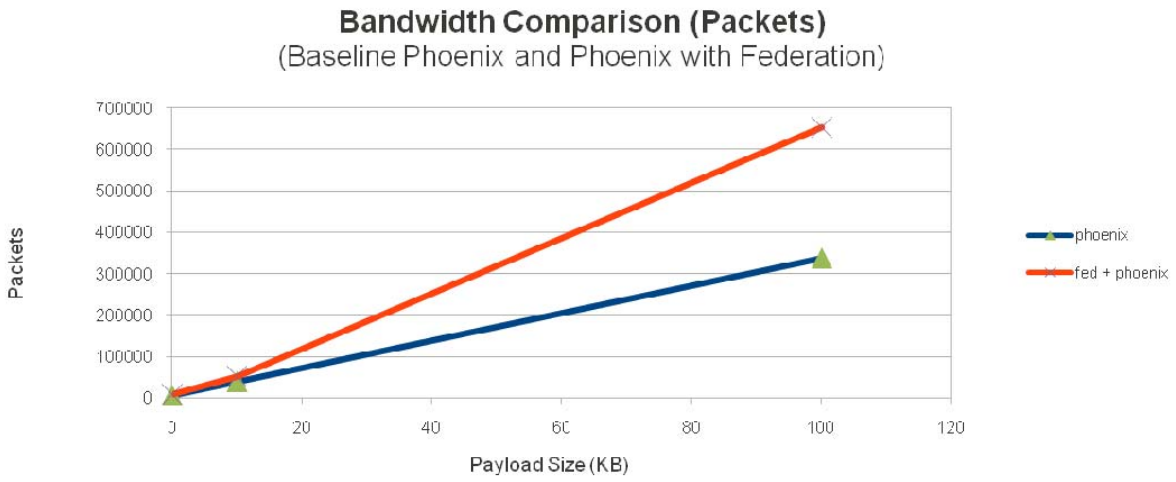

Figure 46: Bandwidth Comparison (Packets) of Baseline Phoenix and Federation

Varying Network Link Capacity

The final set of results we report on is the performance of baseline Phoenix and Phoenix with Federation with different network link capacities. For these particular set of experiments, we used the capability of the NOMADS Testbed to vary the channel capacity from 1024 Kbps down to 56 Kbps. The following tables show the performance results, and the following graphs compare the performance between the two configurations. For these experiments, the publisher was configured to always publish 1250 objects each with a payload size of 10 KB.

Table 20: Performance (Time) of Baseline Phoenix and Federation with Different Channel Capacities

CAPACITY (Kbps)	Performance (Time in Seconds)					
	PHOENIX			FEDERATION + PHOENIX		
	PUBLISHER	1st SUBSCRIBER	2nd SUBSCRIBER	PUBLISHER	1st SUBSCRIBER	2nd SUBSCRIBER
1048	3.634	18.475	18.442	3.224	23.825	23.601
512	286.830	351.732	387.964	544.295	680.855	652.305
256	623.319	820.475	871.529	1264.900	1480.841	1482.407
128	1151.644	1689.977	1809.624	1583.198	3449.844	3399.522
56	2774.609	4394.293	3533.210	5504.411	8152.371	8536.092

Table 21: Performance (Rate) of Baseline Phoenix and Federation with Different Channel Capacities

CAPACITY (Kbps)	Performance (Rate in Objects per Second)					
	PHOENIX			FEDERATION + PHOENIX		
	PUBLISHER	1st SUBSCRIBER	2nd SUBSCRIBER	PUBLISHER	1st SUBSCRIBER	2nd SUBSCRIBER
1048	350.853	69.012	69.136	395.471	53.515	54.023
512	4.445	3.625	3.286	2.342	1.873	1.955
256	2.046	1.554	1.463	1.008	0.861	0.860
128	1.107	0.754	0.705	0.805	0.370	0.375
56	0.460	0.290	0.361	0.232	0.156	0.149

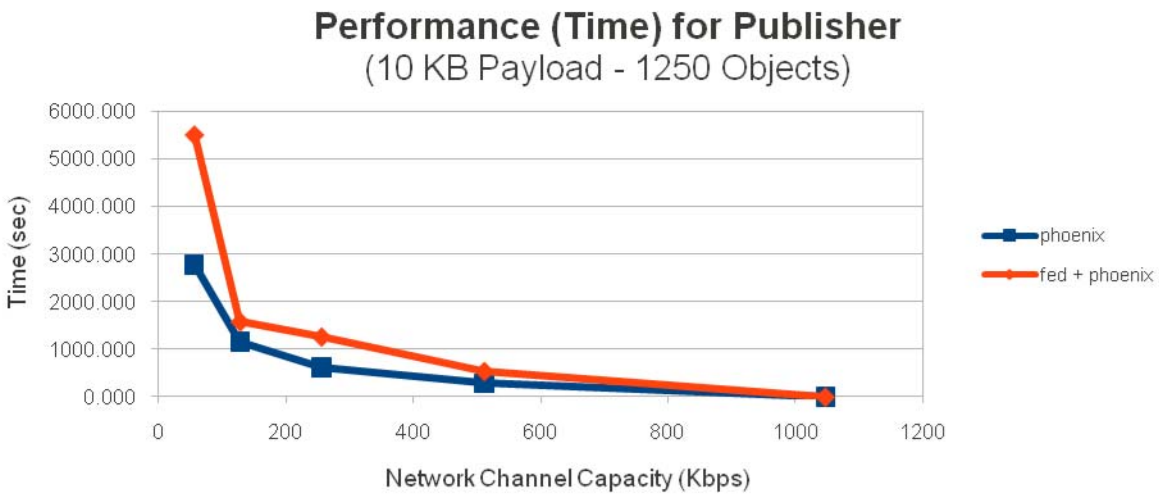

Figure 47: Performance (Time) Comparison for Publisher with Varying Channel Capacities

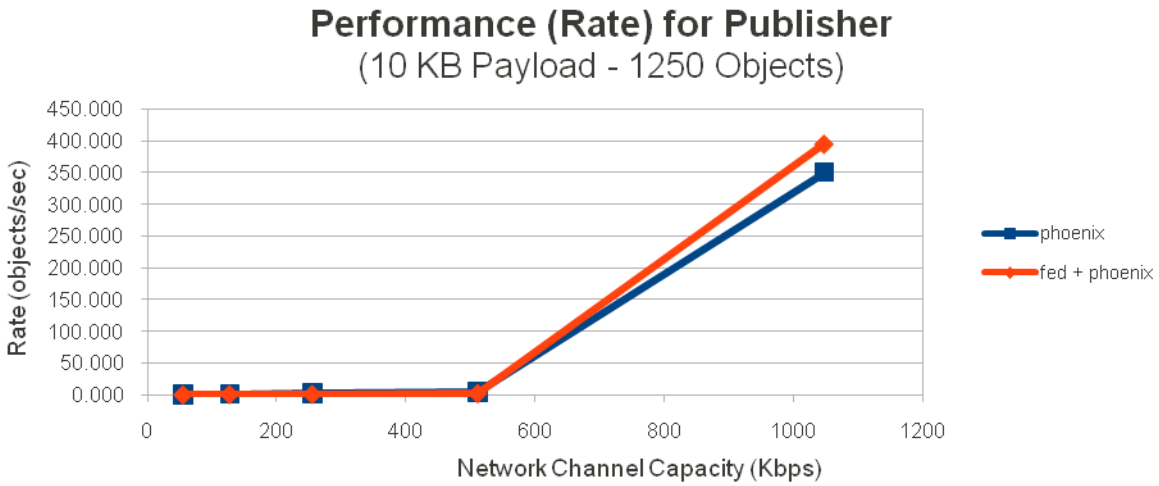

Figure 48: Performance (Rate) Comparison for Publisher with Varying Channel Capacities

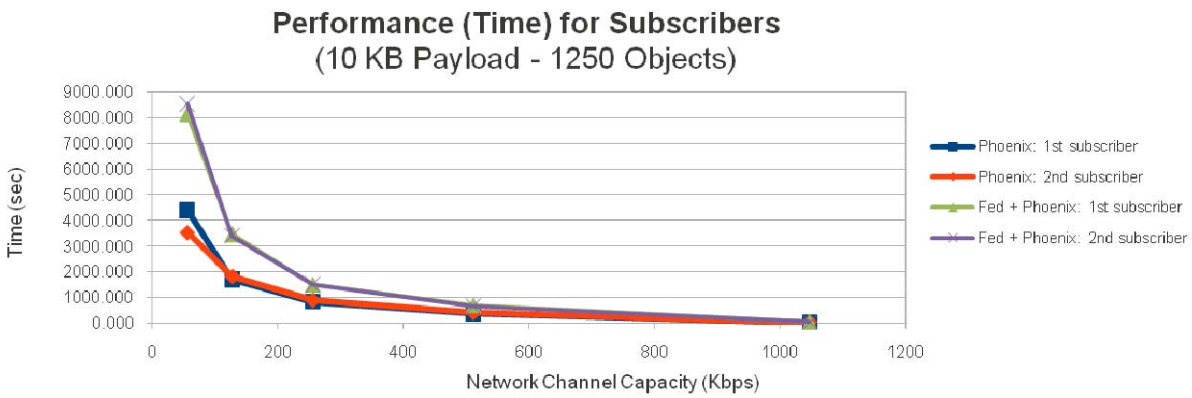

Figure 49: Performance (Time) Comparison for Subscribers with Varying Channel Capacities

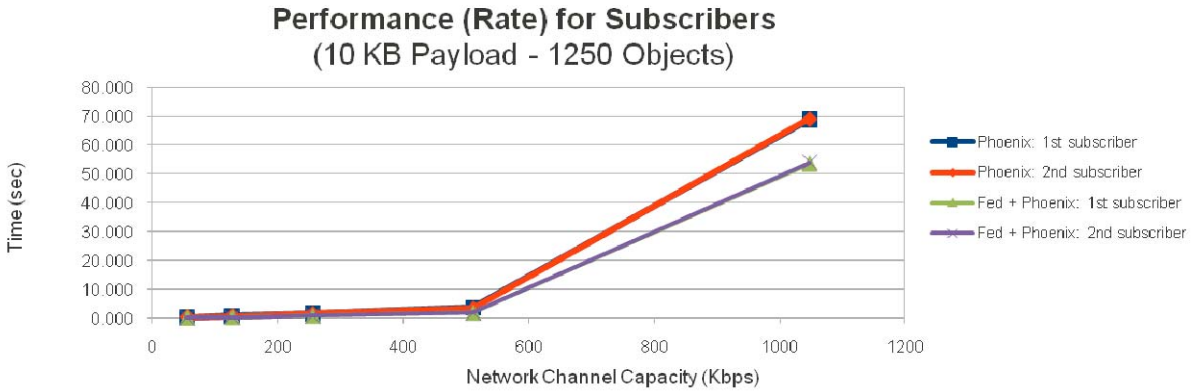

Figure 50: Performance (Rate) Comparison for Subscribers with Varying Channel Capacities

These results demonstrate that the behavior of the Phoenix and Phoenix with federation is as expected under varying network capacities. The time to publish information increases as the network capacity degrades. Also as it is to be expected, the rate of publication and the rate at which objects are delivered to the subscribers increases as the network capacity increases.

4. Conclusions

This project has demonstrated both the importance and the viability of federating multiple information spaces in a policy controlled manner. During the iterative development process, we have designed interfaces and services appropriate for providing federation as well as implemented these capabilities with Apollo, Mercury, and Phoenix. We have integrated KAoS Policy and Domain services components, the XLayer Discovery and Monitoring services, as well as an Adaptation service designed specifically for federation. Extensive experimental analysis has measured the performance of the federation components as well as the integration of federation capabilities into Apollo and Phoenix.

5. Recommendations

While we have demonstrated the feasibility and benefits of Federation, further research and development remains to be accomplished in order to enhance the federation capability as well as

ensure complete integration into the Phoenix implementations such as Fawkes. Below are some suggestion directions for continued research:

5.1 Extensions to Current Federation Capabilities

Our current set of Federation Services provides full support for subscription and publication across the established federation as well as basic federation query implementation. The Federation Manager component enables discovery of new peer federates, dynamic creation of agreements, and fine-grained management of the federation through policies.

While the current approach supports peer-to-peer federation, hierarchical and chain federation structures are also needed for some situations. The Federation Manager could be extended with capabilities to allow the straightforward creation of such federations under policy control, and with the mechanisms needed to route subscriptions, publications, and queries. Moreover, the current federation discovery mechanism can be supplemented with the ability to advertise and discover information needs and offers, as well as exploit knowledge about information requirements to dynamically propose new federations to satisfy them.

The basic federation query could be augmented with a capability for distributed, joint queries and exploitation of intra-query parallelism. Based on a cost model, it could be optimized for parameters such as response time, network utilization, and memory usage by exploiting methods to reduce communication costs and to implement caching and replication of data.

Policy-based publisher and subscriber control could be extended to the federation, allowing producers to advertise capabilities, consumers to request information, and to provide feedback or retraction of inaccurate information *(probably using the Notification Channel in Phoenix*).

A group of federates may not necessarily share common meta-information types. Federation Services could allow the manager to create structural and semantic mappings between metadata-types and then exploit such mappings for data mediation. In addition, autonomic mapping should be incorporated so that the Federation Services can use the anticipated Information Catalog of the IMS with a new, richer, information model.

5.2 Integration with Other Systems and Frameworks

In addition to providing a set of services that enable federating Phoenix, it would also be useful to support other legacy systems and architectures. For example, the addition of CoT Listener and CoT Emitter services could enable seamless integration with existing, non-Phoenix based CoT clients and routers. The addition of these services would bring all of the capabilities of Federation Services, including dynamic adaptation, forwarding, and policy-based control, to handling of legacy CoT-only clients and routers.

At a generic level, federation can be seen as a mechanism to integrate multiple information systems, such as databases (each of which can be viewed as an information enclave), and not just information management systems. Federation should be generalized to provide a policy-controlled approach to transparently and dynamically sharing information across enclaves.

Another important integration should be with Quality-of-Service (QoS) frameworks, such as QED, that are being integrated into Phoenix. The key challenge here is to extend QoS management and enforcement across federates, as opposed to within just a single instance of Phoenix. Example applications such as NCET (Network Centric Exploitation and Targeting) require that information be prioritized across multiple federates, in order to enable timely information fusion across federates. Mechanisms to interface with QoS frameworks should be developed, as well as provide the enforcement mechanisms within the federation capability.

The federation capability relates to ongoing efforts in Cross-Domain information management, currently being developed by AFRL's CDIS group. There are similarities that are worth recognizing, supporting, and exploiting. For example, the CDIS effort performs information shaping for security purposes. Federation performs information shaping for resource management purposes. Both efforts provide transparency to the client, which connects to one and only one instance of Phoenix. Federation should support the efforts of the CDIS group by recognizing common capabilities and leveraging them to avoid duplication.

Finally, the federation services will also interface with system-wide monitoring services, and mechanisms to support survivability. To this end, the federation services will be designed to allow multiple instances to be instantiated to support failover and load-balancing.

5.3 Enhancement of Transport and Dissemination Channels

The Mockets Communications Library provides higher-performance while communicating over wireless ad-hoc networks. The capabilities of Mockets have been integrated into Phoenix as a communications channel. However, Mockets currently does not support asynchronous I/O. Mockets should be extended by implementing asynchronous I/O in order to support asynchronous communication channels within Phoenix.

In addition, DisService provides disruption tolerant, reliable, point-to-multipoint communications. DisService should be integrated as a point-to-multipoint channel within Phoenix. DisService provides a number of advanced features, such as opportunistic listening and multi-channel communications, which could significantly enhance the performance of Phoenix.

5.4 Integration, Evaluation, and Experimentation

The current version of the federation services exist as a branch of the Phoenix repository. However, Phoenix has undergone additional changes after the last iteration of Federation redesign and development. The new features of the federation services need to be integrated into Phoenix on a permanent basis. Furthermore, unit tests need to be developed to provide better support for automated integration testing, as changes continue to be made to both Phoenix and the federation services.

Additional, larger scale experimentation needs to be performed as well, for example on the 96-node NOMADS testbed. The testbed can support medium scale configurations with three-six federates and several clients attached to each federate.

Finally, participation in other experimentation, such as the Limited Technology Experiments (LTEs) that AFRL conducts in collaboration with the Navy, can provide invaluable feedback regarding the success and behavior of federation.

6. References

[1] Infospherics Web Site. Online reference: http://www.infospherics.org.

[2] Grant, R., Combs, C., Hanna, J., Lipa, B., Reilly, J. "Phoenix: SOA based information management services," Proceedings of the 2009 SPIE Defense Transformation and Net-Centric Systems Conference, Orlando, Fl, April 2009.

[3] Linderman, M., et. al., "A Reference Model for Information Management to Support Coalition Information Sharing Needs", In Proceedings of 10th International Command and Control Research and Technology Symposium, 2005.

[4] Suri, N., Rebeschini, M., Breedy, M., Carvalho, M., and Arguedas, M. Resource and Service Discovery in Wireless Ad-Hoc Networks with Agile Computing. In Proceedings of the 2006 IEEE Military Communications Conference (MILCOM 2006), October 2006, Washington, D.C.

[5] Suri, N., Benincasa, G., Tortonesi, M., Stefanelli, C., Kovach, J., Winkler, R., Kohler, R., Hanna, J., Pochet, L., and Watson, S.C. Peer-to-Peer Communications for Tactical Environments: Observations, Requirements, and Experiences. In IEEE Communications Magazine, Vol. 48, No. 10 (October 2010), pp. 60-69.

[6] Loyal J. P., Carvalho, M., Martignoni III A., Schmidt, D., Sinclair, A., Gillen, M., Edmonson J., Bunch, L., Corman, D. QoS Enabled Dissemination of Managed Information Objects in a Publish – Subscribe – Query Information Broker. In Proceedings of the SPIE Conference on Defense Transformation and Net-Centric Systems 2009.

[7] Carvalho, M., Suri, N., Arguedas, M., Rebeschini, M., and Breedy, M. A Cross-Layer Communications Framework for Tactical Environments. In Proceedings of the 2006 IEEE Military Communications Conference (MILCOM 2006), October 2006, Washington, D.C.

[8] Netty Framework Web Site: http://www.jboss.org/netty.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

AFRL	Air Force Research Laboratory
AIMS	Advanced Information Management System
API	Application Programming Interface
CAPI	Client API (Application Programming Interface)
CDIS	Cross Domain Information Sharing
CoT	Cursor-on-Target
IMS	Information Management System
JBI	Joint Battlespace Infosphere
JEFX	Joint Expeditionary Force Experiment
KAoS	Not an acronym – the name assigned to a system
NCET	Network Centric Exploitation and Targeting
NOMADS	Not an acronym – the name assigned to a system
QoS	Quality of Service
SoA	Service-oriented Architecture